UCRL-53751

Converting Scientific Software to Multiprocessors: A Case Study

Robert Eugene Strout II

(M.S. Thesis)

June 20, 1986

Lawrence Livermore National Laboratory

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

UCRL-53751
Distribution Category UC-32

Converting Scientific Software to Multiprocessors: A Case Study

Robert Eugene Strout II

(M.S. Thesis)

Manuscript date: June 20, 1986

LAWRENCE LIVERMORE NATIONAL LABORATORY
University of California • Livermore, California • 94550

Converting Scientific Software to Multiprocessors: A Case Study

By

ROBERT EUGENE STROUT II
B.A. (University of California, Berkeley) 1981

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computing Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

James R. McGraw

Alan C Hindmarsh

[illegible]

Committee in Charge

Converting Scientific Software to Multiprocessors: A Case Study

Abstract

The growing number of multiple processor machines has spurred interest in their immediate use. This paper examines the process and problems involved in converting an application to make use of these machines. A set of Fortran routines forming an ordinary differential equation solving software package is used as the subject of the work. A set of explicit primitives for concurrent programming, the Cray multitasking primitives, is used as the method for exploiting concurrency on a Cray X-MP/48 supercomputer. We first discuss data analysis as a useful technique in the conversion, and examine the tools available within our supercomputing environment for their usefulness in performing the data analysis. Two conversions are performed on the software package. First, to allow multiple problems to execute concurrently, and second, to exploit parallelism within each individual problem. The problems involved in each of these conversions are presented with their considered solutions. Performance measurements are presented for each conversion performed. We find the data analysis procedure too complex and time consuming to perform without appropriate tools. More work than expected was required to produce the first conversion. Finally, the largest source of problems is the lack of sufficient support for multiprocessing from the Fortran language.

Table of Contents

SECTION 1

Introduction and Motivation

This paper examines the process of converting existing scientific software to a multiple processor machine. Many of the applied sciences need greater computational power than exists today. Multiple processor machines provide one way to get that power and they are becoming more abundant in the commercial market. These two factors have led to research into various methods of exploiting concurrency in scientific applications. However, some of the applied science areas need to take advantage of increases immediately. For this reason, we look at the most readily available of these methods: explicit concurrent programming. We intend to look at problems encountered in explicitly converting existing software, and also to determine which tools available to us can aid in the conversion process.

An immediate need exists for greater computational power, greater than can be supplied by today's largest and fastest vector supercomputers. Many current numerical programs require hundreds of hours on these machines. Long delays in producing the results can render a calculation useless by delivering answers that are outdated. Weather prediction programs are classic examples. A tornado prediction for Monday would be useless if the prediction were delivered Tuesday. Other numerical programs simply take too long to justify the answers they would produce.

The new parallel processor machines hold promise for large speedups by allowing programs to take advantage of concurrency generally found in numerical techniques. Sequential architecture computers, on the other hand, are nearing physical limits which will prohibit them from supplying the magnitude increases needed. To help satisfy the need for greater computational power scientists will have to make use of the parallel processor machines. However, scientists who want to take advantage of the new machines with existing programs will find the benefits may not be automatically rendered simply by moving the old program to the new machines.

Many approaches for exploiting concurrency in numerical software are being studied. The more prominent among these include research into the design of concurrent languages[Alla85,McGr83,DiKl85,McKW84,Brin75], the automatic detection of and program restructuring for concurrency[Kuck84,ABKP86,ApMc85], and the use of primitives for explicit concurrent programming[Cray85,FrJS85]. Much of the work done in the prior two areas has not been made available outside the research community. Hence, in order to take advantage of concurrency on a commercial multiple processor machine in the immediate future, we are left with explicit concurrent programming.

A scientific environment in which supercomputers are available has a special concern which must be considered[Zimm85]. That is, large stores of scientific software exist that will benefit from parallel processing. It is common to find very large software packages with hundreds of man-years invested in their development. A programmer developing the software today may very well not be an original developer of the package and may lack an intimate knowledge of the software structure and design. Since automatic detection tools, restructuring tools, and concurrent languages are not readily available, especially for uncommon machines such as supercomputers, the programmer must make explicit changes to exploit concurrency. Two choices avail themselves: make modifications to exploit concurrency in the existing software, or develop new parallel algorithms. In this paper we consider only the conversion of existing software. Both choices, however, represent a large investment of time and manpower to exploit concurrency.

Explicit concurrent programming has been shown to be difficult[McAx84]. The simplest of errors can lead to irreproducible problems that are difficult, if not impossible, to find. However, it appears that our short term goals lay along this path. The work performed here is an attempt to smooth the way for others who must convert existing scientific software to multiprocessors.

This paper examines the conversion of existing software to parallel processor machines. In particular, an ordinary differential equation solver software package is used. Two approaches are considered: (1) a minimal conversion whereby the software may be used concurrently, and (2) a conversion to take advantage of concurrency internal to the package. The problems encountered are described and the attempted solutions are presented. In addition, a comparison is made between the performance versus the conversion effort required for each approach.

The work here is performed in a typical supercomputing environment. The multiple processor machine for which the work is targeted is a Cray-X/MP48[Cray84]. The Network Livermore Time Sharing System, NLTSS, is a locally developed system for supercomputer class machines[DuBo84]. The interface for explicit concurrent programming is the Cray multitasking primitive subroutines[Cray85]. As with most scientific facilities, the predominant language in use is Fortran. By working within this environment we should have the same benefits and limitations as would others.

The ordinary differential equation solver software package used for this study is called LSODE (Livermore Solver for Ordinary Differential Equations)[Hind82]. It is a

Fortran subroutine package that solves systems of ordinary differential equations. The package is very heavily used locally and is also in use at many sites around the country. It has been written to be portable and conforms to the ANSI Fortran 66 standard.

We selected LSODE for this study because it represents a realistic example of scientific software. Since it is heavily used by many of the major local numerical programs, it is a natural candidate for the conversion to multiprocessing. The size of the package is small enough to be manageable (approximately 1500 source lines) and yet presents a reasonable vehicle through which the conversion problems can be studied. In addition, LSODE is a set of library routines intended to be invoked from within an application program. Thus, there are unknown environmental conditions in which LSODE must operate. This helps to bring out problems which may otherwise be suppressed by the fixed, possibly well behaved, environment of a single application program.

We will approach the work in the following manner. First a data analysis will be performed on LSODE. This will supply us with the dependency information required by the following steps. Next we will perform the minimal conversion to produce a version that will support concurrent invocations. This step will concentrate on performing the least amount of work necessary to obtain such a version. The next step will examine the exploitation of concurrency within the LSODE package. During this step, we perform internal restructuring to take advantage of concurrency. No effort is devoted to development of new parallel algorithms .

The remainder of the paper includes the following: Section 2 contains *Background and Terminology* needed to understand LSODE and the synchronization mechanisms used here; Section 3 defines *Data Analysis* is, describes the tools used to perform it, and what is learned by its use; Section 4 describes the *Minimal Conversion Approach* and the concerns addressed in producing a package that can be executed concurrently; Section 5 reviews *Getting an Acceptable Package* from the work produced in the prior section; Section 6 contains the *Internal Parallelism Approach* by addressing problems associated with exploiting parallelism within the software package; Section 7 provides a summary of the *Direct Results* obtained from all the previous sections; and Section 8 presents the *Conclusions* and recommendations for future research.

SECTION 2

Background and Terminology

This section presents background information helpful in understanding the work presented in the rest of this paper. The multiprocessing capability available for this work is first described by a brief characterization of the multiprocessor machine and operating system upon which the work is performed. The primitives available for explicit concurrent programming are then introduced. Since our Fortran compiler allows data scopes not normally found in the Fortran language, we highlight these differences as well. This is followed by an overview of the work allocation techniques used in this paper. Finally, we present an overview of the LSODE package and highlight the portions of it relevant to later discussion.

2.1. The Cray X/MP and the NLTSS Operating System

The multiple processor machine used for this work is a Cray-X/MP48 supercomputer[Cray84]. This machine has four homogeneous processors. Each processor is a pipelined vector processor similar in architecture and performance to the predecessor Cray-1 supercomputer[Cray80]. The individual processors are *tightly coupled*, i.e. they completely share a large common memory of 64 megabytes (8 megawords). In addition, specialized hardware supplies both limited register communication between the processors and a binary semaphore implementation for processor synchronization. These features constitute the concurrent processing capability of the machine.

The NLTSS operating system[DuBo84] supports both distributed and concurrent processing on the Cray-X/MP48. NLTSS is an instance of a distributed message passing/time-sharing operating system for Cray and future supercomputers. It is currently under development at Lawrence Livermore National Laboratory. Concurrent processing is supported by allowing the creation of multiple processes which share the same code and data space in memory. A collection of processes which share memory is called a *family*. Once a family is created, the operating system handles all the basics of scheduling. Parts of the system understand both the family and individual process concepts. However, the process scheduler for time-sharing is now oriented towards individual processes. Each process is treated essentially equally with little consideration of the family concept. This means that although a multiprocessing program may comprise multiple processes that are eligible to run concurrently, the process scheduler may not schedule them as such. All measurements presented in this paper will circumvent this treatment by running the programs on a dedicated system with no schedulable processes other than system processes and members of the family.

2.2. Introduction to the Cray Multitasking Primitives

The explicit concurrent programming primitives used in this work are known as the Cray multitasking primitives[Cray85]. These primitives were designed by Cray Research and implement the *light-weight/heavy-weight* model of computation[Nels81a]. In this model, *heavy-weight processes* correspond to the complete state of a computation schedulable by the operating system. We have previously referred to these as simply *processes* and will continue to do so. The *light-weight processes* are library-controlled threads of execution within the program. These threads are defined by the programmer via the primitives. We will refer to these threads as *tasks* from this point forward. When processes execute in a processor, the execution thread of the process is part of a defined task. If the task reaches a point where it must halt, either to terminate or to wait, the process running the task may switch to another task that is eligible to run. The task switch takes places without intervention from the operating system. It is completely controlled by the multitasking library. On a supercomputer like the Cray X/MP, the cost of operating system intervention is very large. Also, whenever a process switches to the system, it gives up the processor, which may then be assigned to another process. By allowing task switching to take place outside the operating system, a process can continue executing any tasks (i.e. work) that are eligible to run, at a significantly reduced cost. Hence, the number of voluntary process interruptions is reduced in the course of the computation. Multiprocessing is applied to this model by having the multitasking software primitives schedule tasks among a family of processes. This allows multiple tasks to be executed concurrently.

The multitasking primitives allow explicit concurrent programming in terms of tasks and hide much of the machine specific details of multiprocessing. The multitasking primitives allow the creation, deletion and synchronization of tasks. The programmer uses them to define tasks that may be concurrently executed. The multitasking software primitives takes care of all the details of assigning tasks to processes and the switching between tasks. The library software makes use of the special hardware for multiprocessing mentioned earlier, thus relieving the user from having to deal with multiprocessing at the machine and system levels.

The Cray multitasking primitives contain the functionality to create, destroy, synchronize, and test the status of tasks. Figure 2.1 displays the interfaces to the most common multitasking primitives. These interfaces were designed for the Fortran language as can be seen by the interface to the task creation routine, TSKSTART. This routine causes the creation and eligibility for execution of a task starting at the subroutine

```
Task routines:
                    call tskstart ( taskinfo, tasksub, arg1, arg2, ... )
                    call tskwait ( taskinfo )
                    call tsktest  ( taskinfo )
                    call tskvalue ( value )

Lock routines:
                    call lockasgn ( lock, value )
                    call lockon      ( lock )
                    call lockoff     ( lock )
                    call locktest    ( lock )
                    call lockrel     ( lock )

Event routines:
                    call evasgn ( event, value )
                    call evwait    ( event )
                    call evpost    ( event )
                    call evclear ( event )
                    call evtest    ( event )
                    call evrel      ( event )

Barrier routines:
                    call barasgn ( barrier, taskcount )
                    call barsync ( barrier )
                    call barrel      ( barrier )
```

Figure 2.1. Cray multitasking interface with the local extensions for barriers.

tasksub, otherwise known as the *taskhead* of the created task. The routine returns an identifier in the **taskinfo** array which may later be used for TSKWAIT and TSKTEST as described below. The remaining arguments are optional and any number may be supplied. These will be used as the argument list to the subroutine *tasksub* when the task starts and are passed by reference. These optional arguments and their variability of type prevents this interface from being used in languages with type checking. A created task terminates when it performs a RETURN from *tasksub* or the program is halted. The remaining task specific routines are TSKVALUE, TSKWAIT and TSKTEST. The TSKVALUE routine returns the value of a single word supplied at task creation as part of the *taskinfo* array. The TSKWAIT routine causes the task which invoked it to wait until the task specified by the *taskinfo* identifier completes. TSKTEST allows a task to inquire the completion status of another task.

Five primitives compose a suite of *lock* (or binary semaphore[Deit83]) routines. A lock is a synchronization primitive that allows only one task to *own* it at any given time. Locks may be used to protect regions of code by allowing a single task access to the region. These areas of code are normally referred to as *critical sections*, or *single threaded*. A single lock may be used to protect multiple critical sections, thus allowing a single task access to any of them at a given time. The first primitive is LOCKASGN which initializes a lock to *open*. An open lock is one which is not owned by a task. Once a lock has been initialized, a task may request the lock via the LOCKON routine. If another task already owns the lock, the requesting task will wait for its turn to own the lock. If the lock is already open, the requesting task becomes the owner. When the task owns the lock, it is allowed to proceed past the LOCKON invocation. A task may open a lock by calling the LOCKOFF routine. This causes the lock to open and one of the waiting tasks to become its owner. A task invoking LOCKOFF does not wait. The last two lock primitives are LOCKREL and LOCKTEST. The LOCKTEST routine can be used to test the state of a lock, i.e. either opened or closed. LOCKREL is used to release the lock variable. Once a lock has been released, it must be re-initialized with LOCKASGN before it may be used again.

There is a similar set of routines for the *event* synchronization primitive. An event is initialized with the EVASGN routine to an initial state of *clear*. A task may then wait for an event to happen by waiting on the associated event variable with the EVWAIT routine. If the event is clear when EVWAIT is invoked, the task waits until the event is *posted* by another task with the EVPOST routine. Upon posting the event, all tasks waiting on the event are allowed to proceed beyond the EVWAIT invocation which made them wait. If a task tries to wait on an event that is already posted, the task is allowed to proceed immediately. The EVTEST and EVREL routines are similar to LOCKTEST and LOCKREL, respectively.

A *barrier* synchronization primitive is available as a local extension to the Cray multitasking primitives. The BARASGN routine is used to initialize the barrier to *closed,* and specifies the number of tasks that are needed to satisfy the barrier condition. A task which calls the BARSYNC routine for a barrier will wait until **n** tasks, including itself, have reached the barrier (by calling the same routine). The size for **n** is specified at barrier initialization. Once **n** tasks have reached the barrier, all **n** tasks are allowed to proceed and the barrier is immediately closed. The BARREL routine is supplied to release the barrier.

2.3. Data Scopes and Multiprocessing

Many implementations of the Fortran language provide only two data scopes and usually have statically allocated variables. Statically allocated data means that the data resides at a fixed location within the program data space. The two scopes provided are *intra-procedural* data, which are available only within the scope of the routine that declared the variable, and *inter-procedural* data, which are shareable between procedures. The COMMON block construct allows inter-procedural data to be shared among routines that contain the COMMON declaration. The only other method of sharing data between procedures is via the argument passing mechanism, which is *pass by reference*. All other data are *local* variables with intra-procedural scope.

Static allocation of local data variables is inadequate for implementing concurrent processing. Since tasks may execute concurrently and may have common routines in their call chains, an independent set of local variables must exist for each active invocation of a routine. For this reason the local Fortran compiler now supports a stack mechanism for local variables. The space for local variables is dynamically assigned at invocation of the routine, and released when the routine completes. This insures a separate set of locals for each invocation, which are not shared between tasks calling the same routine. The SAVE construct can be used to specify variables that must have static allocation. These variables are shared by all invocations of the routine.

The data scopes available in our multiprocessing environment have been extended slightly. First, local variables are only available within the procedure that declared them. This is standard for all Fortran compilers. However, an independent set exists for each separate invocation of the procedure. Second, SAVEd variables are available within the procedure that declared them. They are shared by all invocations of the procedure. COMMON block variables are available to each procedure that includes the COMMON declaration. These variables are shared by all invocations of these procedures. Finally, the TASK COMMON block construct[Cray85,Cray86] is supported. A TASK COMMON block is similar to a COMMON block except it is dynamically allocated at task creation and is released upon task completion. This has the effect of having an independent set of data for each task declaring the TASK COMMON block. The data is shared by the procedures within the task, but it is not shared between tasks. Figure 2.2 contains a summary table of the scope and lifetime of each of the variable types available.

SCOPE: / TYPE:	Intra-procedural	Inter-procedural Intra-task	Inter-procedural Inter-task
Local/Stack	Procedure (dynamic)	—	—
Saved	Job (static)	—	—
Common	—	—	Job (static)
Task Common	—	Task (dynamic)	—

Figure 2.2. Scope and lifetime for variables types with local Fortran.

2.4. Allocating Work to Tasks

Two approaches to allocating portions of the work to tasks are considered in this paper. The first method is *fixed block allocation*. In this approach, data that may be concurrently operated upon are divided into equal size partitions with one partition for each participating task. All participating tasks are usually required to synchronize with each other upon completion of their portion of the work. This method has the advantage of being straightforward and easy to implement. On the other hand, it has the disadvantage that the equal size data partitions may require unequal amounts of work and thus cause periods of processor inactivity. The imbalanced work load then reduces the amount of parallelism achieved. In addition, if the partitions are large and a task is removed from a processor by the system while all others continue, a similar imbalance may occur.

An imbalance in the work load is better handled by the *self-scheduling* method. In this approach, the data is divided into a pool of small partitions. Participating tasks may obtain a portion of work from the pool and upon its completion it is eligible to obtain another portion from the pool. Each task proceeds in this manner until all the portions have been completed at which point they again synchronize. The imbalance due to either the removal of a task from a processor or the unequal work associated with each partition is reduced for two reasons: the smaller partitions cause the amount of work associated with each to be smaller, and a task with a partition requiring a larger amount of work than normal will overlap with tasks working on multiple small partitions.

2.5. Introduction to LSODE

LSODE[Hind82] is the ordinary differential equation (ODE) solver software package used for this study. It is a subroutine package written with the ANSI Fortran 66 standard. LSODE is a general purpose package for solving initial value problems for explicit ODE systems. Both stiff and nonstiff problems are solved with linear multistep methods[Hind72]. The implicit Adams multistep methods are used for nonstiff systems, and the backward differentiation formulas (BDF) implemented by Gear[Gear71] are used for stiff systems. When stiff options are selected, LINPACK[DBMS79] routines are used to solve the linear systems. Predictor-Corrector formulations of the methods are used throughout, with an attempt to minimize the time needed to obtain the solution. For an overview of the ODE techniques mentioned here refer to reference [Horn75].

A few aspects of LSODE will be the subject of later discussion. First, it should be noted that the overall structure of LSODE is that of the linear multistep methods it implements. These methods are indeed sequential at their highest levels and as such leave few options for parallelism. Exploiting parallelism at this level would require a complete redevelopment of the basic algorithm and is beyond the scope of this work. The individual steps in the basic algorithm are more likely to display a level of parallelism within this scope.

The interface to LSODE includes a problem state variable, method selection, and user-supplied work spaces. The problem state variable, ISTATE, is used for two-way communication between the user and LSODE. The user tells LSODE of the start or continuation of a problem, and LSODE tells the user of successful or unsuccessful completion upon return. The user specifies the selection of the appropriate method, i.e. BDF or Adams, for the problem. The user-supplied work spaces are used internally by LSODE but are also used to contain optional input and output information. The optional information is essentially passed by value, whereas the arguments appearing as actual arguments are passed by reference.

The user is required to supply a subroutine as part of the interface. The subroutine computes the vector function f(y,t), given y and t, for the system of ODEs. We will refer to this subroutine as the *ODE function*. The function is used by the predictor-corrector iterations. Another subroutine, known as the *Jacobian function*, may have to be supplied depending on the method selected by the user. Selection of the BDF method for stiff problems requires a Jacobian matrix involved in the algebraic system to be solved. Two options are available: users may supply a subroutine that computes a value of the matrix at any given y and t, or they may let LSODE compute the Jacobian approximated by difference quotients. In this latter case, the ODE function to compute f(y,t) is used in the difference quotient approximation of the Jacobian.

An auxiliary routine, INTDY, is supplied as part of the LSODE package. This routine is user-callable and returns the k'th derivative of the solution. The user generally makes a call to LSODE to solve an ODE system. He may then call INTDY as many times as desired to obtain various derivatives of the solution to the ODE system. LSODE makes use of the internal problem state COMMON block to obtain the necessary information to calculate the derivatives.

The last aspect of LSODE which concerns this work is the error handling package. The error package is small and has a simple structure. It is compatible with and based upon the SLATEC Error Handling Package written at Sandia National Laboratories.[Jone78] The LSODE error package consists of three routines: the main error reporting routine, and two auxiliary routines to allow changing of the error message control data. The main error reporting routine is callable only by LSODE. The auxiliary routines are user-callable. These routines tailor the error reporting performed by the main error routine. Users can optionally suppress the printed error messages and can change the output unit number to which printed messages are sent (if selected).

SECTION 3

Data Analysis

Data analysis is a technique by which the data dependencies within a program or program segment may be determined. The data dependency information allows us to determine whether or not program segments are independent of one another. We need this information to help us identify which program segments within LSODE may be amenable to concurrent execution.

This section reviews the data analysis process as performed on LSODE. The reasons for performing a data analysis, and a description of the tools and techniques used to perform this data analysis are summarized. The applicability of the available tools to the data analysis process is also reviewed. We conclude this section by highlighting the data dependencies that prohibit the concurrent execution of LSODE, that are brought to light by the data analysis. In addition, we comment on the advantages and disadvantages of the data analysis procedure.

3.1. Introduction to Data Analysis

A systematic data analysis produces data dependency information needed to help determine which sections of a program are amenable to concurrent execution. Many data dependencies are obvious and do not require a systematic approach to identify them. Others may be less obvious or even hidden by the features of the language. For example, large code sections between the data usages involved in the dependencies may be difficult to spot. Because of their separation, these dependencies are likely to be missed in a data analysis of a specific code section. Once code segments can be shown to be independent of each other, i.e. the data they use for input and output does not effect the execution of the other, they become possible candidates for concurrent execution. A simpler definition of *independent* in this context is that no data dependency exists between the code segments. Additionally, once the dependencies between code segments are understood, the segments may possibly be modified to become independent.

The data analysis is helpful to a person who must convert software that was written by someone else. As the conversion proceeds, the data analysis is a useful reference to resolve data dependency questions. In its absence, the dependency issues are generally resolved on a per code segment basis. This type of data analysis can easily overlook dependencies that may lead to incorrect results or behavior, during concurrent execution, which can be irreproducible or extremely difficult to debug[McAx84].

Dependencies are the foremost information kept during the data analysis. Many different forms of data dependencies exist[KKPL81,Lars84], however, we will not distinguish all of them in this work. We limit the notion of *data dependency* as: a dependency exists between an **assignment**(write) and a **usage**(read) of the same data location, or between **assignments** of the same data location. Simple examples of data dependencies can be found in the code segment of Figure 3.1. In this example, there exists a dependency between the usage of **x(i)** at line 1.3 and the assignment to **x(i)** at line 1.4 because the usage must occur before the assignment. A dependency also exists between the usage of **x** in line 1.4 at iteration **i** and the assignment to **x** in line 1.4 at iteration **i-1** since the assignment must occur before the usage.

```
1.1          DIMENSION x(100), y(100)
1.2          DO 100 i = 2,100
1.3               y(i) = x(i)
1.4               x(i) = x(i-1)
1.5    100   CONTINUE
```

Figure 3.1. Simple example of data dependencies in a code segment

A complete data analysis requires an expansion of the scope for dependencies. The simple example in Figure 3.1 shows only those dependencies that can be determined from the limited scope supplied. Dependencies may, however, extend between code segments and even between separate procedures. For example, a dependency exists between the usage of **x** in line 1.4 at iteration **i=2** and the value of **x(1)** which is set outside the scope of the example. The value for **x(1)** may be assigned in a much earlier section of code or in another procedure entirely. We call this an *incomplete dependency* because we do not yet know where the other part of the dependency is found. As a result it is necessary to perform a complete data analysis on each procedure to determine all dependencies internal to the procedure, and also identify incomplete dependencies that cross procedure boundaries.

A data analysis produces various types of information. For each procedure analyzed, we keep track of the *assignment* and *usage* of each variable in the procedure. This effectively determines all internal dependencies for the procedure. In the case of arrays, it is also useful to keep track of the portions of the array used. This will later help to determine independent sections of code that use the exclusive parts of the same array. To identify the incomplete dependencies for the procedure, we must first determine which data may be accessible outside the procedure. Sharing data between procedures is allowed in Fortran by the language features: COMMON blocks, procedure formal and actual arguments, equivalences, procedural parameters, and pointers (if allowed) are a few of them. Once identified, any usage or assignment to such a variable makes it a candidate for a incomplete dependency. A static call chain for the program may then be used to help match incomplete dependencies between different procedures.

There are two types of data analysis. The above discussion has assumed a static data analysis, which assumes that any procedure may be called before or concurrently with the procedure being analyzed. This is not a realistic assumption for most software. With this type of analysis, a worst case set of dependencies is produced. A static analysis can be difficult because of the many possible combinations of data usage and assignment. On the other hand, a dynamic analysis consists of studying the data usage within the control flow of the program and only considering those dependencies that may result. This may reduce the number of dependencies produced by the analysis, but also complicates the analysis by introducing more variability. The control flow through a single procedure may be very complex. Coupling this with the control flow external to the procedure magnifies the complexity even more.

3.2. Tools Available to Aid Data Analysis

The tools available in our environment to aid the data analysis procedure include: FOCAL, SC, and the CIVIC compiler. FOCAL[Coop82] is a FOrtran Code Analysis Listing generator. This utility takes a Fortran source program and produces a cross reference listing of all the symbols (which generally correspond to variables) in the program with the line numbers at which they appear. It also recognizes the assignment of a variable and flags the occurrence in the cross reference as such. This information helps identify assignments and usages of data without having to visually scan the program. A short example program and the cross reference produced are shown in Figure 3.2. In this

```
1              SUBROUTINE matexch ( x, y, n )
2              REAL      x(n), y(n), t
3              INTEGER  n, i
4      C
5              DO 100  i = 1,n
6                 t    = x(i)
7                 x(i) = y(i)
8                 y(i) = t
9         100  CONTINUE
10     C
11             RETURN
12             END
13     C
14             SUBROUTINE matadd ( x, y, z, n )
15             REAL      x(n), y(n), z(n), t
16             INTEGER  n, i
17     C
18             DO 200  i = 1,n
19                t    = x(i) + y(i)
20                z(i) = t
21        200  CONTINUE
22     C
23             RETURN
24             END


1              5    18
100            5    9L
200            18   21L
CONTINUE       9    21
DO             5    18
END            12   24
INTEGER        3    16
REAL           2    15
RETURN         11   23
SUBROUTI       1    14
i              3    5=    6     7     7     8     16    18=   19    19    20
matadd         14
matexch        1
n              1    2     2     3     5     14    15    15    15    16    18
t              2    6=    8     15    19=   20
x              1    2     6     7=    14    15    19
y              1    2     7     8=    14    15    19
z              14   15    20=
```

Figure 3.2. Example of FOCAL output for a simple program

example, the two subroutines making up the program are first listed with reference line numbers and then followed by the generated cross reference. The cross reference section consists of the symbol followed by the line numbers at which it appears. Special markers are appended to line numbers to denote the occurrence of an assignment to the symbol or the appearance of the symbol as a label. For example, the variable **t** is referenced as being assigned to (via the appended equal sign) at lines 6 and 19, while it also appears at lines 2,8,15 and 20. Similarly, the symbol **200** is referenced as being used as a label (via the appended **L**) at line 21, and also appears in line 18.

SC[OHai82] is a simple Structure Chart generator. It takes information produced by another local utility about externals to each procedure. From this information it constructs a graphical representation of all or a portion of the static call-tree beginning at any routine. The output is very similar to a structure chart, except common routines are shown as separate nodes in the call-tree for each reference made from different procedures. The SC utility also allows selective masking of any set of routines. This tool is useful in producing a static call chain to help get a quick overview of the program structure. A limited example of the output produced for LSODE is shown in Figure 3.3. In this example, a structure chart was requested for the subroutine LSODE. The lines leading from LSODE to the other routines denote that these routines appear as calls inside of LSODE. From this, we see a call chain which would allow LSODE to call the routine SOLSY, which in turn would call the routine SGESL, which would then call SDOT. This chart does not tell us when the routines will be called or how often. It does, however, give us an overview of the program structure and where the individual routines fit.

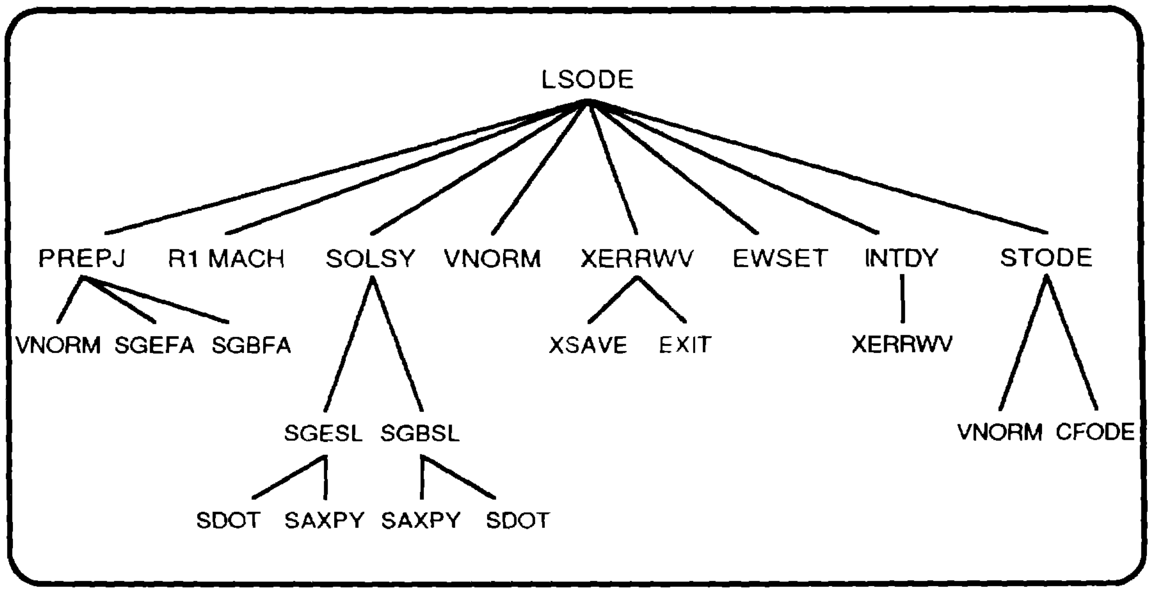
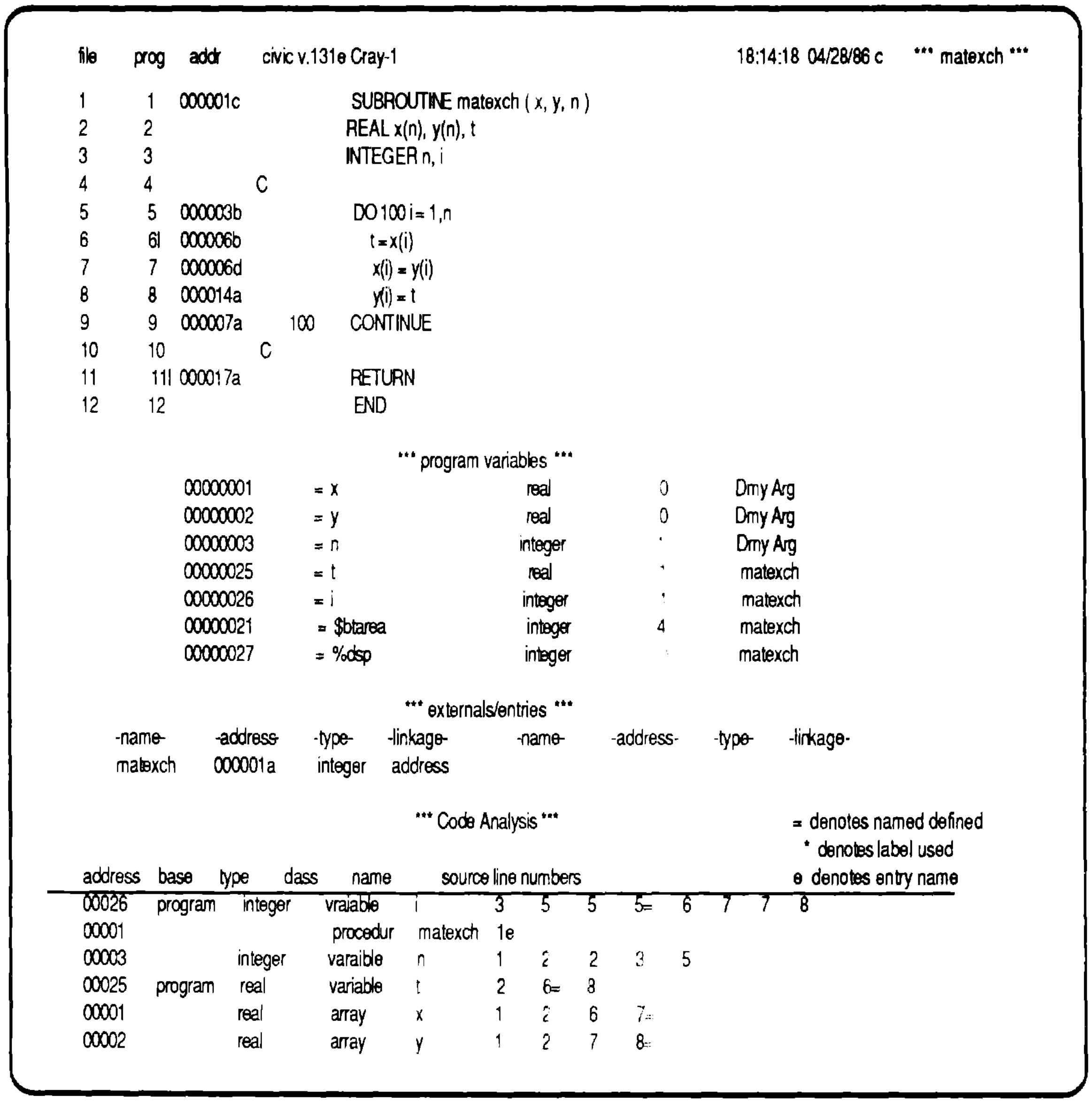


Figure 3.3. Example of SC output for the LSODE software

CIVIC[DeEM82] is a locally supported compiler for the Lrltran language (an extended Fortran) on Cray computers. This compiler supports a code analysis option that can produce a cross reference for each procedure compiled. The cross reference includes all variables, their type, the lines in which they are used or assigned, where they are located (i.e. locally, COMMON blocks, or arguments), and where any procedure invocations are made. The CIVIC listing also includes an indication, for each variable, of whether the compiler was able to determine that it was ever assigned. The compiler does not include interprocedural analysis and so must assume an assignment takes place when in doubt. A typical case where the determination fails is the use of a variable as an argument to a procedure. The CIVIC listing for the example code of Figure 3.2 is shown in Figure 3.4.a. and 3.4.b. Both figures show the original procedure listed with reference line numbers, followed by a description of the program variables and externals, and concluded with a cross reference table. In the **program variables** section, each variable is listed along with its type, size and storage class. The compiler generated variables are also listed here. Storage class refers to the last column in this section. For example, the variable **t** is located in **matexch** and, hence, is local to this procedure, whereas the variable **x** is a **Dmy Arg** (i.e. a formal argument) and so its exact location is not known statically. If a variable were part of a COMMON block, the name of the COMMON block would appear here. Finally, notice that an equal sign appears to the left of each variable name. This tag is an indication that the variable is assigned or that the compiler was unable to determine that the variable is not assigned. In the absence of the tag, the variable has been determined to not be assigned during this routine. The code analysis section is very similar to the FOCAL produced information. The main difference is the inclusion of only the variables and procedures used by this routine.



```
file   prog   addr      civic v.131e Cray-1                              18:14:18 04/28/86 c    *** matexch ***

1      1    000001c              SUBROUTINE matexch ( x, y, n )
2      2                         REAL x(n), y(n), t
3      3                         INTEGER n, i
4      4              C
5      5    000003b               DO 100 i = 1,n
6      6l   000006b                 t = x(i)
7      7    000006d                  x(i) = y(i)
8      8    000014a                  y(i) = t
9      9    000007a       100    CONTINUE
10     10             C
11     11l  000017a              RETURN
12     12                        END

                              *** program variables ***
             00000001      = x                  real          0      Dmy Arg
             00000002      = y                  real          0      Dmy Arg
             00000003      = n                  integer       '      Dmy Arg
             00000025      = t                  real          '      matexch
             00000026      = i                  integer       '      matexch
             00000021      = $btarea            integer       4      matexch
             00000027      = %dsp               integer              matexch

                              *** externals/entries ***
   -name-     -address-   -type-   -linkage-       -name-      -address-    -type-    -linkage-
   matexch    000001a     integer  address

                              *** Code Analysis ***                          = denotes named defined
                                                                              * denotes label used
address  base     type      dass      name       source line numbers         e denotes entry name
00026    program  integer   vraiable  i          3    5    5    5=   6    7    7    8
00001                       procedur  matexch    1e
00003             integer   varaible  n          1    2    2    3    5
00025    program  real      variable  t          2    6=   8
00001             real      array     x          1    2    6    7=
00002             real      array     y          1    2    7    8=
```

Figure 3.4.a. CIVIC listing for example program from Figure 3.2

```
file   prog  addr      civic v.131e Cray-1                          18:14:18 04/28/86 c    *** matadd ***

14     2  000001c            SUBROUTINE matadd ( x, y, z, n )
15     3                     REAL x(n), y(n), z(n), t
16     4                     INTEGER n, i
17     5            C
18     6  000003b            DO 200 i = 1,n
19     71 000006b              t = x(i) + y(i)
20     8  000007c              z(i) = t
21     9  000010d       200  CONTINUE
22     10           C
23     111 000017a           RETURN
24     12                    END

                              *** program variables ***
            00000001       = x              real         0        Dmy Arg
            00000002       = y              real         0        Dmy Arg
            00000003       = z              real         0        Dmy Arg
            00000004       = n              integer      1        Dmy Arg
            00000025       = t              real         1        matadd
            00000026       = i              integer      1        matadd
            00000021       = $btarea        integer      4        matadd
            00000027       = %dsp           integer      1        matadd

                               *** externals/entries ***
  -name-     -address-  -type-   -linkage-      -name-     -address-    -type-    -linkage-
  matadd     000001a    integer  address

                               *** Code Analysis ***                    = denotes named defined
                                                                         * denotes label used
address  base     type     dass      name     source line numbers       e denotes entry name
00026    program  integer  vraiable  i        4    6=   6    6    7    7    8
00001                      procedur  matadd   2e
00004             integer  varaible  n        2    3    3    3    4    6
00025    program  real     variable  t        3    7=   8
00001             real     array     x        2    3    7
00002             real     array     y        2    3    7
00003             real     array     z        2    3    8=
```

Figure 3.4.b. CIVIC listing for example program from Figure 3.2 (continued)

3.3. Data Analysis Problems and the Usefulness of the Tools

Each of these tools were applied to the data analysis of LSODE with varying degrees of success. A description of how the tools are applied and the problems found is given. Problems in performing the data analysis are highlighted. We then review the usefulness of the available tools to the data analysis problem.

A systematic approach to performing the data analysis was chosen. The SC tool was used to produce the static call-tree for LSODE and the FOCAL tool used to produce a cross reference. The information produced allowed a more rapid analysis of individual procedures. Starting the per-procedure analysis at the leaf procedures in the call-tree was convenient as these routine tended to have a simple structure. Once all leaf procedures are analyzed, we have a set of internal dependencies associated with each procedure and a common set of incomplete dependencies. Other information noted for each routine is similar to that mentioned earlier, i.e., the type, storage class, usage and assignment of all variables. In addition the partial usage of arrays was noted. Care was taken in identifying incomplete dependencies, since usages and assignments to COMMON block or pointer variables are not always obvious.

After all leaf procedures are analyzed, we may include interprocedural analysis into the approach. We next proceed up the call-tree one level to the set of all unanalyzed routines that have all their lower (i.e. called) routines analyzed. This set of routines are analyzed as the leaf routines were and, in addition, we try to complete any incomplete dependencies by looking at methods for sharing data with the routines already analyzed (e.g. COMMON, saved, TASK COMMON, pointer, equivalence, and argument variables, as well as shared i/o). We continue in this manner until all routines have been completed. Upon completion, we have a set of internal dependencies for each procedure, and a common set of global dependencies between procedures.

Adjustments were made to the data analysis procedure shortly after starting. First, the FOCAL tool was abandoned in favor of the CIVIC compiler listing. The FOCAL listing did not distinguish between local variables in different procedures that had the same name. Thus the information concerning assignment to a variable had to be verified to see which routine it was in. As an example, look at the variable **t** in Figure 3.2. FOCAL is also unable to distinguish between appearances as usages and appearances in declarations. This prohibits us from rapidly telling whether the variable is actually used. On the other hand, the CIVIC listing supplied most of the information needed for the analysis. Second, we realized that for the multitasking method being used to exploit the parallelism it is only necessary to consider the interprocedural dependencies. The final goal of this work, however, is to exploit parallelism internal to LSODE. When this stage is reached, the internal dependencies are needed for procedures in which parallelism is to be exploited. The information is useful when breaking a procedure into separate procedures in order to exploit the internal parallelism. Rather than determine the internal dependencies for every procedure, we postpone the activity until the procedures with candidate internal parallelism have been identified.

Language related problems hindered the data analysis procedure. Fortran supports pointer and equivalence variables, as well as procedure passing. These features tend to hide dependencies and thus require more attention during the analysis. For example, a pointer may cause references to data in different COMMON blocks even though the pointer variable itself is local to the procedure and the COMMON blocks are not declared by the procedure. This creates a dependencies between the usage of the pointer variable and the other usages of the data in the COMMON blocks. However, these features are supported in most Fortran implementations and hence their presence must be accepted.

An example of an analyzed leaf procedure from LSODE is given in Figure 3.5.a. The variable usages leading to incomplete dependencies are listed in Figure 3.5.b. Since the amount of information rapidly becomes unwieldy, variables involved in dependencies and their usage are also summarized at the top of this figure. The usage, **assigned** or

```
5.1           SUBROUTINE xsave ( mflag, lun, ifun )
5.2     C
5.3           INTEGER mflag, lun, ifun, mesflg, lunit
5.4           COMMON /ls0001/ mesflg, lunit
5.5           DATA  mesflg/1/, lunit/6/
5.6     C
5.7           IF ( ifun .EQ. 0 ) then
5.8              mflag = mesflg
5.9              lun = lunit
5.10          ELSEIF ( ifun .EQ. 1 ) then
5.11             mesflg = mflag
5.12          ELSEIF ( ifun .EQ. 2 ) then
5.13             lunit = lun
5.14          ENDIF
5.15    C
5.16          RETURN
5.17          END
```

Figure 3.5.a.   Leaf routine from LSODE.

```
SUMMARY:

              XSAVE ( mflag,      Assigned/Used
                      lun,        A / U
                      ifun        U

              /ls0001/ mesflg,    A / U
                       lunit      A / U
_____________________________________________________________

Incomplete Dependencies:

Variable      Scope        Usage        Lines

IFUN          argument     Used         5.7,  5.10,  5.13

if IFUN=0:
  MFLAG       argument     Assigned     5.8
  LUN         argument     Assigned     5.9
  MESFLG      common       Used         5.8
  LUNIT       common       Used         5.9

if IFUN=1:
  MESFLG      common       Assigned     5.11
  MFLAG       argument     Used         5.11

if IFUN=2:
  LUNIT       common       Assigned     5.13
  LUN         argument     Used         5.13
```

Figure 3.5.b.   XSAVE incomplete dependencies. Dynamically, either 3 or 5 exist.

used, is useful in quickly determining whether to consider the dependency. If trying to match a **used** variable, only dependencies involving assignment need be considered.

The tools for data analysis must be an integrated set. The bookkeeping for this analysis was done by hand, with most information kept at the procedure level. The amount of information for a single procedure can be great and the task of bookkeeping for all procedures overwhelming. An integrated set of tools that created and maintained a database of the information would reduce, if not remove, the bookkeeping task.

Each tool used in the data analysis provided some useful information but also lacked facility to provide more information where it could. FOCAL recognized assignments to variables and provided a way to rapidly find variable references. However, it lacked the ability to distinguish between variables with the same name in different procedures. This inability made the recognition of assignments practically useless for our purposes. SC allowed a quick overview of the code structure or sections of the code structure. This was very valuable because there was no previous familiarity with the program. On the other hand, SC lacked to ability (or information) to detect procedural arguments and include them in the structure chart. When not aware of this, the structure charts can be misleading. Finally, the CIVIC listing provided almost all the information required to perform the analysis. It distinguishes between subroutines, functions, and variables making it easier to detect function invocations than with the FOCAL output. Yet as close as CIVIC is in producing all the needed information, it still fell short in places. It did not distinguish between variables for which it could and could not determine assignment. In cases where the compiler is unable to do so, we must make the determination manually. Without the distinction, we must manually intervene in all cases where the compiler declares an assignment. Vectorizing compilers also determine most of the dependencies internal to a procedure, but do not supply a way to have them listed. Additionally, variable usage as an actual argument to a procedure call or invocations of procedural arguments did not appear in the cross reference. Few compilers provide global analysis and so are unable to provide both interprocedural dependencies and exact determination of variable assignment. In either case, Fortran language constructs inhibit their completeness. Of the tools available, the compiler proved to be the most useful in the data analysis procedure.

3.4. What was Learned about LSODE ?

The data analysis identified problems which must be corrected before LSODE can be run concurrently. Some of these problems were obvious from the beginning. Others were not so obvious, and would likely be overlooked by a person (like myself) not familiar with the software package. The following paragraphs briefly describe the problems.

An internal COMMON block is used to maintain state for an LSODE problem. This state is maintained during and across calls to the LSODE package. Auxiliary routines in the LSODE package may be used between calls to LSODE and make use of the problem state maintained in the COMMON block (i.e. the routines in LSODE are COMMON coupled). In addition, routines internal to the package change variables in the COMMON block. Since COMMON blocks are shared data, we are only capable of maintaining the state for one problem at a time. This prohibits concurrent execution of LSODE problems.

The user-supplied ODE function is allowed to use its arguments beyond the scope defined by the LSODE interface. A few of the arguments passed to LSODE are in turn passed to the ODE function. Since arguments in Fortran are passed by reference, it is allowable to pass LSODE an array which is greater in length than LSODE expects. The portion of the array beyond the LSODE definition is used to pass extra information to the user-supplied ODE function. Usage of the extended portion is unknown to LSODE and hence outside its jurisdiction.

The error reporting package in LSODE uses data-loaded COMMON and output devices. The user is allowed to change the status or destination of error messages via auxiliary routines in the package. Initial state is created by data-loading the COMMON block entries that control the error messages. The auxiliary routines change the status or destination information in the COMMON block. If concurrent calls to these routines occur, it is possible to get an inconsistent state in the COMMON block. Additionally, the error package reports errors encountered to an output device. Concurrent error messages sent to a common output device may become intermixed.

Finally, the data analysis identified the arguments to LSODE that are possibly modified. Data for arguments that are not modified by LSODE may be mutually used by concurrent invocations. The arguments which are possibly modified must be supplied data independent of all other concurrent invocations.

3.5. Pros and Cons of the Data Analysis

We now make a few observations on the advantages and disadvantages of the data analysis procedure. First, we found the information produced to be extremely useful, and consider it an invaluable procedure for anyone not intimately familiar with the software. It helped us to avoid trouble with unforeseen data usages. Secondly, we realized that for multitasking method being used to exploit parallelism (i.e. at the procedure level) we need not determine the internal dependencies for all procedures. Only those for which internal parallelism is to be exploited was this necessary. Generally, the interesting information is the interprocedural dependencies, all other data is local to a procedure and therefore isolated from the effects of concurrency. Finally, an in depth data analysis is extremely difficult and time consuming when performed manually. Even for a small and simple code like LSODE, with approximately 1400 lines, it was a burdensome task. For the large codes which must eventually make use of multiprocessors, it will be impossible without the aid of an appropriate tool. Tools using the techniques developed by Kennedy and Kuck will have to become available before such codes may consider performing a complete analysis. The most natural place for this analysis to take place is in a compiler. It does a large part of the analysis already and may be able to take advantage of the extra information.

SECTION 4

Minimal Conversion Approach

This section examines the process of doing a minimal conversion of LSODE to a multiprocessor machine. We define *minimal conversion* as the minimum amount of work necessary to allow concurrent execution of independent LSODE problems. Our motivation for performing a minimal conversion is that it may often be desirable to provide an interim version of software which supports concurrent execution, while a more complete conversion is in the works. The conversion is only needed if it is desirable to run parallel LSODE problems or use LSODE within the framework of larger parallel problems. Producing an interim version of software is a temporary measure. We will impose guidelines upon the work which are felt to be normal and reasonable when producing interim software. These guidelines in their approximate order of importance are: perform the minimum amount of work, avoid changes to the interface, limit the amount of time available for the conversion, and try to maintain portability where possible.

Performance is not an overriding goal for the conversion. Since the conversion will produce a software package capable of running multiple LSODE problems concurrently, it would be expected to get near complete overlap of the problems. Performance measurements at this level have little meaning. The more interesting question is: what happens to the sequential performance between the original package and the new package?

4.1. Review of the Conversion Problems

4.1.1. Stack-based Conversion

The first issue to be addressed in the LSODE conversion is the switch from static Fortran to stack based Fortran. This is a very common issue and is briefly discussed here. The switch is a necessary part of fulfilling the reentrancy requirement for a multiprocessed program. Static Fortran supplies static locations for all local and global variables. This prohibits more than one instance of a procedure from being invoked concurrently because only one set of local variables may exist. In addition, programs developed under such a compiler may make use of the condition and rely upon variable values being retained across procedure invocations. Converting to stack based Fortran solves the first problem and aggravates the second. Each procedure now dynamically allocates new storage for its local variables at each invocation. This allows more than one invocation of a procedure to be made concurrently. However, now that the locals are allocated with each invocation, no guarantee can be made that the same space will be allocated in subsequent calls. This breaks the possibility of value retention.

A test was made to help detect the reliance upon value retention in the LSODE code. A single program ran two problems through LSODE by alternating calls to the LSODE package between them. This should cause the local variables to change between calls for the same problem. After completion it ran the problems separately and then compared the answers. LSODE passed this test. We found later this test was not sufficient. It is likely that a subset of the internal variables between the two problems will have common values. The test does not detect value retention for these variables. The oversight in the test was brought to our attention by a value retention problem which appeared in the stack based LSODE. The problem occurred because the local link utility zeroes the uninitialized data space in the code it produces. Yet, in stack based code it is possible to reach a stack overflow condition which causes the data space to be expanded. This expanded space is not initialized with zeroes. If a new invocation received initialized space for its locals, a variable would have an initial value of zero (in this case a legal value). If the uninitialized space was allocated, the initial values would not be zero (in this case an illegal value). The problem surfaced as a floating arithmetic error which made it easy to find. However, it could just as easily surfaced as a irreproducible incorrect answer that would be very difficult to find.

A tool could have been developed to try to detect the usage error described above. The tool would have to follow all possible paths through a subroutine and detect any usage of local variables before they have been assigned. Developing such a tool is beyond the scope of this work.

The last of the stack based conversion concerns is the use of non-reentrant library routines. In order for LSODE to be considered reentrant, all of its call chain must be safe from concurrency problems. The support library routines called are not reentrant and are outside the control of LSODE. The only quick solution available until the libraries become reentrant is to single thread access to each complete library. This is done via a global lock placed around each call to such a library routine. However, for this to work, the user of LSODE must make similar use of the same lock around his calls to routines in this library. This changes the interface slightly until reentrant libraries are available.

4.1.2. Problem State Common Block

The second conversion issue is the internal state kept by LSODE in a COMMON block. This problem was described in Section 3.4. To allow concurrent execution of independent LSODE problems, we must supply a method by which the internal state may be kept on a per problem basis. Four options are considered as methods by which this may be accomplished. We label these methods as (1) *Task Common*, (2) *Move to User Space*, (3) *Move to User Space with Local Save and Restore*, and (4) *Pointered Common Extension.*

The *Task Common* method replaces the COMMON block with a TASK COMMON block. The switch of the COMMON declaration to a TASK COMMON declaration is the only change involved. This option is the easiest to implement, and it also maintains the LSODE syntactic interface. However, since TASK COMMON blocks do not retain their values across task invocations, the following usage scenario is prohibited. The user may decide to step a problem (via consecutive calls to LSODE) by making each call a separate task. This makes LSODE the taskhead for each of the tasks. Under this usage, the internal state will be lost upon return from the initial task and, therefore, unavailable to the subsequent tasks.

The *Move to User Space* method replaces the internal COMMON block with work space provided by the user. The LSODE syntactic interface must be changed slightly to require a larger work space be provided. In addition, each internal routine and auxiliary routine that needs access to the state information must have the work space passed to it through the argument lists of its call chain. This means internal and auxiliary interfaces must be changed to account for the work space. Each of these routines will then be forced to reference the needed information via indexed elements into the work array. The internal changes involved to use this method can be great, and the prospects for maintaining such a code are disheartening. Yet, this method allows the usage scenario prohibited by the *Task Common* method described above. The disadvantage is that the user is now made responsible for providing and maintaining the independent work spaces during and between invocations for the same problem. This creates a possible source of errors against which LSODE can not protect.

The *Move to User Space with Local Save and Restore* option is a combination of the previous two methods. This method involves changing the COMMON block to a TASK COMMON block, and also requires the user to supply an enlarged work space. Upon appropriate entry and exit of LSODE, the TASK COMMON block is restored from the work space and saved to the work space, respectively. This has some of the same problems as the other two methods. That is, the LSODE interface must change slightly, the user is still responsible for maintaining the work spaces, and some of the user-callable auxiliary routine interfaces must be changed to allow for the work space. Two other considerations exist for this method. First, the cost of saving and restoring the TASK COMMON block is added to each invocation. Second, the user-supplied work space contains out-of-date information while an invocation is active. We do not consider these to be great disadvantages for LSODE. This method does allow the scenario prohibited by the *Task Common* method. Additionally, it minimizes the internal changes to consist only of the switch to TASK COMMON and the save and restore upon entry and exit. The maintenance issue is also improved with this method.

The last option to be reviewed is the *Pointered Common Extension*. This is not really an option since it requires an extension to the Fortran language. It is presented here because it removes two of the problems from the previous method. This method extends Fortran to provide another type of COMMON block, the pointered common block. The user still supplies the enlarged work space. However, upon entry to LSODE the pointered common block is "pointed" at the work space. This allows us to refer to the individual elements of the work space by a meaningful name and also removes the cost of saving and restoring at entry and exit. An example of a possible syntactic form for such a feature is shown in Figure 4.1. The pointered common block name is assigned with the work space argument. Any references from this point on will map to references into the **wrkspace** argument, e.g. **c** is equivalent to **wrkspace(3)**. This facility could also be supplied via a macro preprocessor and Fortran that supports equivalencing to arguments. The disadvantage to this extension is that it causes the same problems in performing a data analysis that the pointer variable now causes. Finally, we do not review this option as a proposal for a language extension, but rather to help emphasize the lack of Fortran language support for this multiprocessing associated problem.

```
SUBROUTINE work ( wrkspace )
POINTER COMMON / pcom / a, b, c, d
pcom = wrkspace
    .
    .
c = a * b
    .
    .
END
```

Figure 4.1. Possible syntax for a *Pointered Common Block*

The *Task Common* option was chosen for this phase of the work. It satisfies three of the four guidelines we placed upon the work at the beginning. That is, it represents the minimum amount of work, can be done in a short amount of time, and maintains the LSODE interface. The only disadvantage is that a usage scenario is prohibited.

4.1.3. Error Package

The final issue to be addressed is the error reporting package found in LSODE. The problems with this package were described earlier in Section 3.4 and a overview of the error package was given in Section 2.5. The problems can be summarized as (1) making use of data-loaded COMMON, and (2) using a common output device. Data-loaded COMMON insures that the initial call to LSODE has the necessary information to enable it to correctly report the error. The information of concern here is the error message status flag, which enables or disables printing of errors to the output device, and the logical unit number, for the output device to which messages are to be printed. The information may be kept globally or on a per problem basis. Global information forces the use of a common output for all LSODE problems. The usage of a common output device demands some form of synchronized access. Without such access, portions of error messages that are reported concurrently may become intermixed. In addition, it would become difficult to tell which problems generated each specific error.

We consider two methods by which the above problems may be addressed. First, the error message control information may be kept on a *Global Basis*. That is, it will effect the error reporting of all LSODE invocations. Second, the control information may be kept on a *Per Problem Basis*. This requires a separate set of control information for each LSODE problem. Both of these options are discussed below. In each case, it is assumed that all Fortran reads and writes handle contention for the device at the level of the logical unit.

Let us first consider keeping the error message control information on a *Global Basis*. This method forces the use of a common output device which aggravates the two problems mentioned above. A solution to identifying the problem that generated the error is trivial. It is solved by a slight modification to the interface to include an optional problem identification string. This string is then printed with the error message. The second problem of intermixed portions of error messages can be solved by synchronization methods. The error reporting package can be single threaded. We do not expect to degrade the efficiency of parallel execution because of the single threaded error package. The error package executes quickly and infrequently. Single threading of the error package is revisited shortly. In addition, the flexibility allowed the user is reduced by forcing all error messages to be sent to a common device. It may be desirable to have errors for separate problems sent to separate output. However, this scheme maintains the syntax and monoprocessing semantics of the error package. That is, a request to change the error message control globally affects the output.

This scheme requires additional synchronization to prevent the error message control information from changing in the middle of its use. It is possible for the user to concurrently change the output device while an invocation of LSODE is active. If the invocation were in the middle of reporting an error, a commitment may already exist for it to report the error to the old device. This may lead to an error if timed appropriately. Assuming the error package is single threaded, the lock used there can also be used by the auxiliary error routines to control the changing of the control information. For example, an invocation of LSODE trying to report an error claims the error package lock so that it may have uninterrupted access to the control data. While this is happening, a concurrent call to an auxiliary error package routine to change part of the control information is made. The auxiliary routine will try to claim the error message lock and be forced to wait until the LSODE invocation is finished reporting the error.

This synchronization is not sufficient to completely protect the error package environment from changing. Limitations upon the way in which the environment can be changed must be placed upon the user. For example, if the user were to close the old output device, call an auxiliary error package routine to signal the change, and then open the new output device all concurrently with an active invocation, two errors might occur. First, the active invocation may be committed to report an error to the old device at the time the device is closed. Second, after the call to the auxiliary routine, the active invocation may become committed to report an error, and attempts to do so before the new output can be opened. There is no method by which LSODE can protect against these external changes other than relying upon the user to follow some limitations. In this case, the limitations are that the user must first open the new output, signal via the auxiliary routine, and then close the old output. Any other order may lead to errors during execution.

Let us now consider keeping the error message control information on a *Per Problem Basis*. Problems are dynamic with respect to the LSODE package. Therefore, the control information will have to be kept in dynamic storage. Up to this point, the problem is exactly the same as the internal state problem addressed earlier. That is, we need more than one set of control variables. The options available for that problem are also available here. Each option comes with the advantages and limitations discussed at that time. Yet, a complication is added here that requires the variables to have an initial value at the first invocation for each LSODE problem. The issue is complicated because not all dynamic forms of variables in Fortran allow data-loading. Therefore explicit coding must be added to initialize the error message control upon the first invocation for each problem, which is difficult to detect in stack based concurrent code. However, this information is available via the LSODE interface.

Other changes may also be required by this method. First, if the TASK COMMON option is not chosen, the interfaces to the auxiliary routines in the error package must be changed to allow identification of the problem reporting the error. This can be easily be performed by merging the error message control information into the previously expanded user-supplied work space. The work space already represents the state of an LSODE problem and can be supplied to the auxiliary error routines. Also, if the local save and restore option is chosen, a change requested by a task on behalf of another task that is active may be lost because the change is made to the workspace and is not seen by the active task. When the active task returns from LSODE, the changed value is overwritten by the restore action. There is no way to update the information in the active task, because we cannot access another task's TASK COMMON blocks.

This method also requires the synchronization scheme and user limitations described for the previous method. It allows the user the flexibility of sending errors to different outputs and avoiding the common output problems mentioned above. However, the user may still specify a common output for all LSODE problems. As long as this can occur, we must provide as much protection as we were able to with the previous method.

Both schemes presented allow common output. While LSODE is able to handle contention for the output file between instances of itself (via the single threading mechanism), we cannot guarantee that the user will not contend for the same file. We have been unable to find a method by which LSODE can completely protect against this occurrence. The two alternatives which present themselves are: share the locking mechanism with the user, or place limitations upon the user's usage of the output file. Sharing the lock with the user allows greater flexibility. The output device is likely to be available most of the time because error reporting is short and infrequent. The low contention for the output allows the user relatively flexible access. Alternatively, we can limit the user's use of the output device. A condition could be set upon the user to not access the output unit while any LSODE invocation is active. This is a rather severe limitation. In either case, we cannot insure correct usage. We must trust the user to obey the restrictions or make correct use of the lock provided.

Keeping the error message control information on a *Global Basis* was selected for this conversion. It provides the least flexibility for the user, but requires the least amount of change to the program and maintains the current interface. We are also influenced by the previous choice of TASK COMMON to handle the internal state problem. If another choice had been made and many of the necessary changes already required, then our decision might have been different.

4.2. Summary of Results from the Conversion

The results obtained from the minimal conversion of the internal state and error package can be summarized as follows. First, Fortran provides little language support to handle problems introduced by concurrency. Many options and trade-offs presented themselves for the internal state problem. None of them were complete, and most were made difficult by lack of language support. Second, the management of input and output devices can be difficult if not impossible, especially when trying to maintain the current interface and semantics. The semantics may not directly translate from a monoprocessing environment to a multiprocessing environment. Also, the synchronization needed to handle contention for the output device in this case was simple but incomplete. As long as there is no guaranteed cooperation to insure that the environment does not change, complete protection cannot be supplied.

The correctness of LSODE cannot be fully protected from the actions of the user. To summarize, we are able to provide protection for the majority of the problems. However, we are unable to provide complete coverage and must ultimately rely upon the user to abide by some condition, e.g. using a shared lock or obeying a usage limitation.

As long as the we rely on the user, the chance for error increases. Such errors will probably be inconsistent and difficult to find.

The LSODE package produced proved to be too limiting. After reviewing the package with the original author, the package was deemed to be too limited and complicated to be of much use. Too many restrictions and conditions for usage were placed upon the user. These added to the complexity of an already nontrivial package. We then decided to continue the conversion to produce a package which removed enough of the conditions to be considered usable by the original author. The limitations of this package and the further conversion are addressed in the next section.

SECTION 5

Getting an Acceptable Package

This section reviews the work performed to convert the LSODE package to a package that is acceptable for use. The minimal conversion effort produced a package that placed many limitations and conditions for usage upon the user. We felt that they made the package unacceptable for use. We briefly introduce the limitations and usage conditions for the package. We then present the changes made to bring the package to a level acceptable for use. Only the changes made are discussed, though many others were considered. Less importance is placed upon the guidelines under which the minimal conversion was performed. These guidelines were unsuccessful in producing a usable product. The section is concluded with a presentation of some simple performance measurements made with the acceptable version.

5.1. Relaxing Limitations

The limitations and conditions on the use of LSODE produced by the minimal conversion are due to the methods selected to solve the problems. For more explanation, than is presented here, refer to Section 4. Figure 5.1 presents a summary of the limitations imposed on the usage of LSODE. The first two limitations are by-products of the selection of TASK COMMON to solve the internal state problem. They are due to the properties of TASK COMMON, i.e. values are not retained across a task's lifetime and TASK COMMON data is not sharable between tasks. The third condition is caused by the lack of reentrant library. The fourth condition is due to the decision to keep error message control information on a global basis. This decision forces the usage of a common output device that becomes the central reason for the fifth through seventh conditions. It was mentioned earlier that the package was already nontrivial to use. The combination of this and the newly imposed limitations, in our opinion, make the package too complicated to use.

<u>MINIMAL CONVERSION LSODE - LIMITATIONS</u>

1. LSODE may not be invoked as a taskhead if either of the following are true:
   a. Any of the LSODE auxiliary routines are to be used upon return from LSODE, or
   b. The problem is to be continued with another call to LSODE.

2. The use of the LSODE auxiliary routines is restricted to the task from which the desired LSODE invocation was made. To avoid this restriction, it is necessary to explicitly transmit the problem state information between tasks. This requires the use of synchronization.

3. Until reentrant libraries are available, the user must use the LSODE error lock around all calls to the FORTLIB library which occur from a multiprocessed section of code. These include Fortran read and write statements (as they are calls to FORTLIB routines). For example,

```
COMMON /er0001/ errlock
INTEGER errlock
CALL LOCKON (errlock)
write ( ... ) ...
CALL LOCKOFF (errlock)
```

4. Errors reported by LSODE cannot be directed to separate files for each problem. Switching error control effects all current and subsequent problems.

5. If you wish to use the same output as currently assigned for LSODE error messages (from a multiprocessed section of code), you must either:
   a. use the LSODE error package lock around any access to the output file (this may already be satisfied by condition 3); for example, see the usage in 3 above; or
   b. refrain from accessing the output file until there are no active invocations of LSODE or the auxiliary routine INTDY. Also, no new invocations of these routines may be made until you are through accessing the output.

6. There are imposed conditions for the usage of the error package auxiliary routines, XSETF and XSETUN, when called from a multiprocessed section of code. They are:
   a. a new output file must be available BEFORE the call XSETUN() or XSETF(1) is made.
   b. the existing output file may be closed only AFTER the call XSETUN or XSETF(0).

7. Invocation of the auxiliary error package routines, XSETF or XSETUN, may cause the calling task to wait. This happens when an active invocation of LSODE or INTDY is printing an error message. The wait helps protect the routine from the loss or modification of the environment as it know it.

Figure 5.1. Limitations on LSODE produced by minimal conversion

The first change is to switch from the *Task Common* option to the *Move to User Space with Save and Restore* option to solve the internal state problem. This method retains the benefit of being able to refer to the state information by name, rather than by index into the work space. This greatly increases readability and future maintenance. This selection also removes restriction #1: not using LSODE as a taskhead. Thus, all usage scenarios are supported. Of course the LSODE interface must be changed slightly to require a larger work space, but that is not a concern for this work.

Switching to this option requires changes to the LSODE auxiliary routines. First, the routines that explicitly save and restore the problem state may be removed. The action of these routines are now automatically performed at each LSODE entry and exit. Second, the auxiliary routine, INTDY, must be changed to allow the workspace to be supplied as part of its argument list. LSODE also makes use of this routine internally. If the interface is changed, then corresponding changes to its call chain must be made internal to LSODE. In addition, the work space would first have to be updated from the TASK COMMON used by LSODE. Rather than make these changes, we opted to separate the routine into a user-callable version, still called INTDY, which depends upon the work space, and an internal version called INTY, which makes use of the TASK COMMON block. This removes restriction #2 because the auxiliary routines have either been removed, or the problem of interest is now specified via its work space.

Three disadvantages still remain with this selection. The cost of performing a save and restore is paid for each LSODE invocation. The user work space is outdated while an active LSODE is working on the associated problem. The user is responsible for the correct use and maintenance of the independent work spaces. We do not considered them to be limitations. Dependence upon the user for correct treatment of the work spaces can easily lead to errors. However, this is a dependence that the original LSODE had.

We would like to remove or ease the remaining restrictions associated with the error reporting package. These restrictions are derived from the choice to keep error message control information on a global basis. Restriction #4 can be removed by moving the control information into the already expanded work space, i.e. using a *Per Problem Basis* solution. The work spaces are per problem data structures which will allow the control information to be kept individually. The complication of initial values for the control information can be resolved as described in Section 4.1.3. Since the user may always specify the same output to all LSODE problems, restriction #5 cannot be removed. However, because of changes described below, this restriction is relaxed a little. Figure 5.2 contains a restatement of the relaxed restriction.

The final two restrictions are due to concurrency problems caused by the presence of the auxiliary error package routines. We decided to remove these routines and merge their functionality into the optional input mechanism already found in LSODE. By merging the mechanisms with the LSODE invocation, it becomes impossible for the error message control information to change while LSODE is active. However, it does not protect against premature closing of the output by the user.

Restriction #5.b can be relaxed a little. The dependence upon LSODE and INTDY occur because both these routines may cause the error package to be invoked. We have modified the user-callable version of INTDY to no longer access the error package. A error value is now returned through the calling sequence. This removes the dependence upon INTDY from this restriction.

We have been able to increase our protection from the actions of the user. The merging of the auxiliary error routines's functionality into LSODE is an example of this. The protection still remains incomplete. We cannot protect against the user making concurrent use of an individual work space. We are also unable to protect against the premature closing of an output device supplied to an LSODE invocation.

The limitations remaining for the acceptable version of LSODE produced by this work are shown in Figure 5.2. The first restriction, locking around calls to a non-reentrant library cannot be removed until a reentrant library is available. The second is less of a restriction than it is a common-sense reminder about the multiprocessed environment. The last restriction remains essentially unchanged from its previous form. Even so, locking around access to the output is not considered to be a major restriction.

<u>ACCEPTABLE CONVERSION LSODE - LIMITATIONS</u>

1. Until reentrant libraries are available, the user must use the LSODE error lock around all calls to the FORTLIB library which occur from a multiprocessed section of code. These include Fortran read and write statements (as they are calls to FORTLIB routines). For example,

```
COMMON /er0001/ errlock
INTEGER errlock
CALL LOCKON (errlock)
write ( ... ) ...
CALL LOCKOFF (errlock)
```

2. An output file must remain open as long as an invocation of LSODE which uses it is active.

3. If you wish to use the same output as currently assigned for LSODE error messages (from a multiprocessed section of code), you must either:
   a. use the LSODE error package lock around any access to the output file (this may already be satisfied by condition 1); for example, see the usage in 3 above; or
   b. refrain from accessing the output file until there are no active invocations of LSODE. Also, no new invocations may be made until you are through accessing the output.

Figure 5.2. Limitations for the acceptable version of LSODE

5.2. Performance Measurements for Acceptable LSODE

The conversion produced a package that allows multiple LSODE problems to be run concurrently. Since the resultant package uses no synchronization, except during error reporting, it is possible to get full overlap between independent problems. Measurements that reflect this are uninteresting. We have chosen instead to measure the performance of the new package on a single problem and compare this to the performance obtained from the original.

A series of test problems were run through both the new and original software. The test problems each exercise a commonly used path through the software and represent realistic examples of typical problems. Each test was run with a range of problem sizes. The questions addressed are: did performance degrade? and if so what was the cause of the degradation?

The overall results from the tests were a small loss of performance incurred by the conversion. The performance degradation ranged from 1% to 9%. The degradation can be broken down into three components: the cost of saving and restoring the work space to the internal TASK COMMON, the cost of using dynamic variables (i.e. TASK COMMON and stack locals), and the cost of stack-based subroutine linkage.

The latter two costs are very closely related. First, the cost of the save and restore operation was small. The problem execution times ranged from 0.5 to 200 seconds. The degradation due to the save and restore ranged from 0.2% to 0.001% of the total problem time, respectively. For very small problem times one will notice a greater relative degradation. Second, their is no noticeable cost of using dynamic variables.

Finally, the cost of stack based subroutine linkage was less than 1% for the LSODE package. Most paths through the software do not make a large number of calls to the user-supplied functions. Those paths that do not often call the supplied function retained the lower degradation. However, the paths that make a large number of invocations experienced a degradation up to the 9% limit observed. However, if the new package is not being used for concurrent processing, it is allowable for a non-stack based function to be supplied. A degradation of less than 1% was then observed.

SECTION 6

Internal Parallelism Approach

Many applications may not be able to gain an advantage from the LSODE produced by the previous conversion efforts. They may only have a single problem to run or may be unable to run their multiple problems in parallel. For these programs, it is desirable to speedup the execution of single problems. Exploiting the parallelism found within the LSODE package is the next obvious step.

There are a number of concerns when exploiting the parallelism within the package. First, the ability to perform concurrent problems obtained from the previous conversion must be maintained. Second, since the package runs on the Cray X/MP-48, we should avoid disrupting the vectorization that already exists. Within any section of code, the maximum speedup of scalar code obtainable through multiprocessing is equal to the number of processors (4). Whereas, vectorization on the X/MP-48 can speedup some code sections by a factor of eight to ten. Even greater speeds are realized with intelligent use of the vectorization capabilities. Finally, if tasks are created internal to LSODE, there is an unknown interaction with the rest of the application program. Either the application or other libraries may also be creating tasks. The tasks we create may interfere with the tasking in these other places.

6.1. Identification of Candidate Areas

Parallelism will be exploited at the lower levels of the LSODE implementation. We mentioned previously that the redesign of the higher levels of LSODE to take advantage of parallelism was beyond the scope of this work. The identification of program areas that are candidates for exploiting parallelism at the lower levels will be approached in two steps. The areas which consume large portions of computation time are identified. Then each of these areas are examined for exploitable parallelism.

6.1.1. Hot Spot Study

A *Hot spot* study was performed to identify areas which may result in a substantial time saving. A *hot spot* is a program area that contributes a large fraction of the total computation time. Many program areas may display parallelism. However, if the program area only contributes a small fraction of the total computation time, exploiting its parallelism will lead to a small reduction of the total time. Many of these cases do not justify the work involved to exploit their parallelism. An attempt to get large gains should be directed toward the hot spot areas. We concentrated the hot spot studies on the most commonly used LSODE modes.

Hot spots are identified with the locally available TIMER/TALLY tool.[Nels81b] The tool takes a statistical sampling of the locations in the program counter register by halting the program at equally spaced times with small time intervals. The resulting data is then post-processed with corresponding symbol table information to produce, for each procedure, a statistical graph of the time spent within various sections of each procedure. A high statistical percentage generally corresponds with an area making a large contribution to the computation time. In addition, the instruction being executed at each sample is noted. From this, a breakdown of the contribution due to scalar versus vector instructions is made. This is useful in determining how well a procedure is vectorized. Also, a per-procedure contribution to the overall computation time is given.

The TIMER/TALLY tool is limited by the statistical sampling methods it uses. Any statistical sample may display spurious peaks where there may indeed be none. Generally, this can be overcome by increasing the size of the sample. However, to increase the sample size you must increase the total computation time; some programs are not able to do this. Another possible source of error is the fixed sampling increment. It is possible that the program being sampled executes in phase with the sample period. This might also cause spurious peaks, as well as the lack of real peaks. These limitations did not effect our measurements as we were able to expand the LSODE problems measured to any reasonable size.

The hot spot study was performed on a set of test problems with a few controlled properties. The test problems were kept large and consisted of systems of 100 or more equations. This was done to help increase the statistical sample for TIMER/TALLY and to insure that the granularity of the problem would be sufficient for the multiprocessing primitives to be used. The five most commonly used modes of LSODE were studied. We shall refer to these modes as they are numbered in Figure 6.1. Also, in order to reduce the dependence on the user problem the ODE functions used in these studies are near minimal cost. This will help focus the hot spot detection on areas within LSODE.

Mode	Characteristics
10 -	Nonstiff systems (Adams method)
21 -	Stiff dense systems (BDF) User-supplied Jacobian routine
22 -	Stiff dense systems (BDF) LSODE computes difference quotient Jacobian
24 -	Stiff banded systems (BDF) User-supplied Jacobian routine
25 -	Stiff banded systems (BDF) LSODE computes difference quotient Jacobian

Figure 6.1. Five most commonly used modes of LSODE

The hot spot study revealed the subroutines that contributed the largest portions of time to the test problems. Figure 6.2 provides a summary of the results. The largest contributions are due to a small number of routines. The user-supplied ODE function is the major contributor for modes 10, 21 and 22. For modes 10 and 21, this usage is completely due to the corrector iteration logic used by the multistep algorithm. For mode 22, the majority (90%) of the total time is due to evaluation of the function in computing the difference quotient Jacobian (DQJ), while 7% is for evaluations due to corrector iterations. The user-supplied Jacobian subroutine also contributes to mode 21. The LINPACK routines for solving systems of equations contribute heavily to modes 24 and 25, and less so to modes 21 and 22. The increase in relative contribution by the system solvers from modes 21 and 22 to modes 24 and 25 is due to the substantial decrease in the number of evaluations needed in the banded case. As the number of evaluations decrease, the system solvers and the LSODE multistep algorithm become more apparent.

	Problem Size (number of equations)		
Mode	100	250	500
10	82% F	91% F	95% F
21	59% F 28% J 11% SysSolv	—	—
22	97(90)% F † 2% SysSolv	—	—
24	50% SysSolv 31% Lsode	54% SysSolv 31% Lsode	55% SysSolv 30% Lsode
25	45% SysSolv 33% Lsode	52% SysSolv 32% Lsode	53% SysSolv 31% Lsode

† 90% due to difference quotient Jacobian calculation

F -	User supplied ODE function
J -	User supplied Jacobian routine
SysSolv -	LINPACK System Solvers
Lsode -	LSODE multistep algorithm

Figure 6.2. Major contributors to total time for each test problem

Three areas present themselves for examination as worthwhile candidates for exploiting parallelism. These are the system solving subroutines, the user-supplied Jacobian routine, the user-supplied ODE function, and the code segments surrounding them. As mentioned before, redesign of the multistep algorithm will not be addressed in this work. Hence, its contribution to modes 24 and 25 will be ignored. The remaining three areas will be examined for parallelism.

6.1.2. Examination for Parallelism

Parallel implementations of the LINPACK system solving subroutines have already been written.[ChDH84] With access to this work, we could easily replace the sequential routines with the parallel ones. A reasonable benefit should be apparent for modes 24 and 25 where the system solvers dominate the total computation time. Rather than repeat this work, we leave the system solvers alone and direct our attention to the other areas.

The multistep algorithm used by LSODE at its highest level is sequential. The time stepping nature of the multistep algorithm is the force that makes the algorithm sequential. Both the corrector iterations and the usage of the Jacobian matrix are bound to this sequential algorithm. For modes 21 through 25, the Jacobian matrix is required at the various time steps. Therefore, the Jacobians must be supplied sequentially. In addition, a single invocation of the Jacobian subroutine supplies the new Jacobian matrix and keeps us from overlapping invocations of the subroutine. For these reasons, we are inhibited from parallelizing the evaluation of the user-supplied Jacobian function in any way. The user may, however, exploit concurrency within the routine.

The user-supplied ODE function contributes heavily to the computation time for modes 10, 21 and 22. The evaluation of the ODE function for modes 10 and 21 is completely attributable to the corrector iteration logic. Its usage is similar to that of the Jacobian subroutine. The corrector iteration logic is sequential and requires a single evaluation of the routine for each iteration. For similar reasons to the above, we are unable to exploit parallelism in this area. However, examination of mode 22 shows that only 7% of the total computation time from evaluations are from the corrector iterations. The largest contribution (90%) to this mode is from the evaluations performed in computing the DQJ. Formation of a single DQJ requires multiple evaluations of the user-supplied ODE function. Our only hope in parallelizing the DQJ formation is in performing these evaluations in parallel with each other. Figure 6.3 shows the program segment that causes the evaluation of the user-supplied ODE function while computing a DQJ for method 22.

```
      j1 = 2
      DO 230 j = 1,n
         yj = y(j)
         r = amax1(srur*abs(yj),r0/ewt(j))
         y(j) = y(j) + r
         fac = -hl0/r
         CALL F ( neq, tn, y, ftem )
         DO 220 i = 1,n
220         wm(i+j1) = (ftem(i) - savf(i)) * fac
         y(j) = yj
         j1 = j1 + n
230   CONTINUE
```

Arguments: y, ewt, neq, ftem, wm, savf

Task Common: n, tn

All others are local variables

Figure 6.3. DQJ formation code segment which calls the user-supplied ODE routine, F

6.2. Modification of a Candidate Area

An examination of the DQJ formation code segment leads to issues that must be addressed in order to exploit the parallelism. The code segment calculates a column of the Jacobian matrix with one evaluation of the user-supplied ODE routine, F, for each iteration of the **230** loop. With the assumption that this loop will be made into a task (and therefore a subroutine), we must make sure that the relocated loop has access to all necessary data. The TASK COMMON variables **n** and **tn** as well as the argument variables **y, ewt, neq, ftem, wm,** and **savf** cannot be obtained in the new subroutine/task via declarations. They, along with the local variables **j1, srur, r0, hl0, fac,** and the routine **F** must be passed to the new subroutine as arguments to insure the necessary environment.

The individual iterations of the loop are not independent of each other but can be made independent. The **y**, **ftem** and **wm** arrays are global (because they are supplied as arguments) and each is assigned during an iteration. Before this loop can be parallelized across its iterations, we must insure that the usage of these arrays do not conflict. The **wm** array is the resultant matrix produced by this loop. Each loop iteration computes an independent column of the matrix. The **y** and **ftem** arrays are not independent across iterations. Each iteration requires the original **y** array with a single element modified. Concurrently executed iterations would cause more than a single element to be changed. The **ftem** array is used as a temporary during each iteration. A new array is computed at the beginning of each iteration and used at the end of the iteration. Concurrent iterations would cause the values for one iteration to be written over by another, or a mixture of the values could result. These problems are easily solved by supplying each task with a separate copy of the **y** and **ftem** arrays. This requires the use of dynamically sized arrays to handle different problems sizes. Fortran does not supply a facility to handle this, but through the use of a memory management package and pointer variables we can create such arrays.

The first attempt to parallelize this section of code is the *straightforward* approach. A fixed block partitioning is used to break the work among the tasks. Since iterations can be performed concurrently with each other, the total number of iterations may be divided into as many groups as there are tasks. The information concerning a given group of iterations is then passed to the task. The easiest way to implement this is to create a new set of tasks for each DQJ to be computed. At the point this loop would normally be computed, we can divide up the work and create the tasks needed. We then wait for all tasks to complete before we continue. The tasking primitives allow us to easily do this through use of the TSKSTART and TSKWAIT primitives. Each task starts by allocating private copies of the **y** and **ftem** arrays and then computing the assigned portion of the matrix. The program segments used to implement this method appear in appendix A.

The straightforward approach causes some concerns to be raised. First, the **y** array used in the original loop is the original array supplied by the user. A feature of LSODE is that this array is handed back to the user-supplied ODE function. Since Fortran uses pass by reference for its arguments, the user was allowed to pass an array which was larger than required by LSODE. This allowed extra information to be passed through LSODE and back to the user function for use. The straightforward approach only copies the known portion of the **y** array for each task, and so the extra information is lost. This is not viewed as a significant loss of functionality. Second, the size of the problem determines the granularity of the parallelism in the loop. For small problems, the overhead of the tasking primitives may be large enough to cause a degradation on the overall compute time. Task creation and termination are the two primitives with the highest cost. Two options were considered for this problem: a mode to force sequential (non-tasking) computation was considered, and avoiding the cost of creation and termination of tasks for each DQJ .

Protection from the user is the last concern. Since the parallelization of the DQJ formation involves concurrent evaluation of the user-supplied ODE function, we must be sure that the routine supplied is reentrant. The use of a non-reentrant routine would be disastrous. Again, we are helpless in providing protection. The only way to protect against the user supplying a non-reentrant routine is to singly-thread access to the routine by placing locks around its invocation. This is pointless as it would inhibit the concurrent evaluation of the function which was the initial motivation for work. A slight complication is that the user may be unaware that LSODE is making concurrent calls to his routine, or he may not know what constitutes a reentrant routine.

The next parallelization method, *Pre-create tasks*, addresses the overhead of the primitives compared to the granularity of the problem size. Task creation and termination are the highest cost primitives. Creating new tasks for each DQJ computation produces the highest overhead possible. Since all tasks are equivalent (i.e. have the same taskhead) and the LSODE problem state variable ISTATE indicates the beginning or continuation of a problem, it is possible to create one set of tasks at the beginning of an LSODE problem and retain them until the problem is complete. Whenever a new DQJ is to be computed, the tasks are signaled and the calculation proceeds. This avoids the overhead of repetitive task creation and termination.

There are problems in implementing the task pre-creation method. First, a synchronization mechanism must be devised that causes the pre-created tasks to wait until it is time to compute the next DQJ, proceed when ready, and wait upon completion. Many attempts to provide such a mechanism were made. Many of these were more elaborate than necessary because they were attempts to use a small number of locks/events. These invariably had a case for which they would not operate. The simpler mechanisms which used a number of locks/events were easier to implement and functioned properly. We present two of these methods in a later discussion.

Allocating work arrays for each task and sharing data among a group of tasks are also problems encountered by pre-creating tasks. Each task need only perform the allocation of their **y** and **ftem** arrays once. This is easily handled at task creation by preallocating an area for the arrays which can be handed to the task through its argument list. Additionally, some of the data needed for the task's calculation won't be available at the time of pre-creation and so cannot be passed as part of the task's argument list. To handle this, the data needed by the task is divided into two categories: (1) the data that remains fixed between task creation and usage by the task, and (2) the data that is not known until just before task execution. The first group of data can be supplied as arguments to the task. The second group, however, must be communicated to the task just before it is signaled for execution. Some of this information must be shared by the tasks that are cooperating on this problem (e.g. lock and event variables), but not shared with tasks which are working on other LSODE problems. This presents a problem as there is no data structure in Fortran that supports this data scope. TASK COMMON shares data only with the procedures in the individual task, while COMMON shares data with procedures in any task. We again get around this deficiency by making use of memory management routines and pointer variables. By allocating a block of memory and loading it with addresses of the data to be shared, we can pass a pointer to the shared memory as an argument to the task at creation time. This is effectively a second pass by reference argument list for the task to use once it has been signaled. It also requires that the task knows the structure of the shared memory block. The location of the shared memory block may be included as part of the problem state kept in the user-supplied work space between invocations of LSODE for the same problem.

The last problem, which cannot be handled, is the possibility of a change of the user-supplied ODE function between calls for the same LSODE problem. With the method described above, the secondary argument list cannot be used to pass the function to the task. This is because there is no way in Fortran to load the address of the procedure into the shared memory block and give the task the ability to invoke the procedure. We are forced to pass it through the task's argument list at creation time. This restricts the user from changing functions between calls for the same LSODE problem. We do not consider this to be a notable restriction as this ability is very rarely, if ever, used.

The first synchronization method used with the task pre-creation scheme is based on *barriers*. At task creation, a barrier is initialized to *n+1*, where *n* is the number of tasks pre-created for this problem. Each task immediately waits on the barrier. When the main task reaches the point where a new DQJ should be computed, it sets up the secondary argument list and signals the waiting tasks by waiting on the barrier. This completes the barrier and all tasks are allowed to proceed. The main task immediately waits upon an event that will be used to signal him when the new matrix is ready. The other tasks continue by picking up the secondary argument list data and computing their assigned portion of the matrix. As each task completes its portion, it enters a critical area and increments a counter for the number of tasks completed, leaves the critical section and again waits on the barrier. The last task to enter the critical region detects he is the last and signals the main task that the work is completed. He then resets the completion counter and exits the critical region to go wait on the barrier. The main task resumes and immediately clears the completion event for the next time through. This mechanism is fairly simple. The program segments for this method can be found in appendix A.

Self scheduling is the next technique used to distribute the data among the tasks. Balancing the work load among the tasks can become a concern if the amount of time to calculate equal portions of data can differ or if there may be multitasking taking place elsewhere in the program. The self-scheduling technique tries to address these concerns. An unequal work load is not a concern in LSODE and the conditions under which performance tests will be run will assure the absence of multitasking elsewhere in the program. This method is performed to look at the cost in the amount of overlapped execution due to implementing self-scheduling with the multitasking primitives. The synchronization mechanism between the main and other tasks is the same as used for the barrier method. However, the work is not divided into pre-assigned groups at task creation. After the tasks have been signaled, each task enters a critical region and grabs the next iteration to be performed, releases the critical region, and goes to compute the single iteration. Each task continues to do this until all iterations have been completed. The program segments which implement this method are shown in appendix A.

Changing the barrier synchronization to *event* based synchronization for signaling the tasks was tried to make a comparison between primitives with differing cost. The event mechanism requires more calls to the event routines than the barrier mechanism makes to the barrier routine. Also, the cost of a call to a event routine is more expensive than a call to a barrier routine. The event mechanism works by initializing an event for each pre-created task. The tasks begin execution and immediately wait in their individual events. When the main tasks reaches the DQJ formation, it individually signals each event and then waits on a completion event as in the previous mechanisms. The tasks proceed by first clearing their individual events and then continuing in the same manner as the previous mechanisms. The results of this conversion are presented in Section 6.3, and the program segments for this method can be found in Appendix A.

6.3. Performance

The performance of the four implementations is presented for the original 100 equation test problem used in the hot spot study. The test is a single problem and all speed increases can be attributed to the exploited parallelism within LSODE. The speedups are measured in terms of the amount of processor overlap obtained during the complete problem. The results of the the performance tests are presented in Figure 6.4.

METHOD	Overall Overlap	Effective DQJ Overlap
Straight Forward	2.78	3.47
Pre-Create Tasks w/ barriers	2.95	3.77
Pre-Create Tasks w/ barriers and Self-Scheduling	2.95	3.77
Pre-Create Tasks w/ events and Self-Scheduling	2.91	3.69

Figure 6.4. Performance results for each implementation for internal parallelism

Initially, the speedups obtained appear somewhat disappointing. The program seems only capable of using approximately three of the four available processors. This performance is actually quite good. The maximum obtainable overlap which can be obtained from a code with serial portions can be calculated as $m = 1/((P/N)+S)$, where m is the maximum overlap, N is the number of processors being used, P is the fraction of total compute time in a sequential run contributed by the area to be multiprocessed, and S is the remaining contribution (i.e. $S=1-P$). Our previous hot spot study provides us with a value of 0.9 for P, and 0.1 for S. The X/MP gives a value of 4 for N. Therefore, a maximum speedup of $m=3.08$ is possible for this problem. In this light, the actual speedups obtained appear more satisfying.

It is interesting to determine the overlap obtained by the multiprocessed section. By rewriting the previous equation for computing the maximum overlap as $N = P/((1/m)-S)$ and using the actual overlap measured, we can estimate the effective number of processors used by the multiprocessed section. The effective processor usage for each of the four implementations is included in Figure 6.4.

The work involved to obtain these results seems reasonable. The first method was very simple to implement and gave us a good speedup. The following methods which involved pre-creation of the tasks gained us a little more. However, these last three methods involved a rather messy implementation for the pre-creation, and even messier implementation for the passing of data to the tasks. The work involved to get this little extra speedup seems unjustified for a general purpose package. The work would be justifiable if the mechanisms for pre-creation of tasks and passing data to them could be made available in a clean manner. Large problem sizes which cause the small gain in overlap to become a large gain in compute time would also justify the work.

There are two other interesting results from the performance figures. First, the change from fixed partitioning to self-scheduling did not change the overlap. Since the conditions that self-scheduling covers did not exist we would expect any overlap change to be attributable to the addition of the self-scheduling synchronization. However, there appeared to be no noticeable effect due to the extra synchronization. This is due to the cost of the extra synchronization overlapping with the computation by the other tasks. The second result is in the conversion from a barrier to an event based synchronization for task signaling. The signaling mechanism is a sequential operation and does not overlap with the other tasks. The use of the more expensive event primitive caused the loss of a tenth of a processor in the multiprocessed section.

6.4. Other Types of Candidate Areas

The types of parallelism addressed so far have been restricted to the low level parts of the LSODE multistep algorithm. We have examined each low level part for internal parallelism. Though this work does not include the redevelopment of the multistep algorithm, we did examine the possibility of rearranging the parts of the multistep algorithm to try to exploit parallelism. As an example, a new DQJ computation could start immediately after the completion of a DQJ computation. This would allow calculation of the new DQJ to overlap the usage of the one just computed. This (overlapping) type of parallelism is apparent only at the higher levels of LSODE. The hot spot study does not help in its identification. With explicit techniques, it may only be identified by a complete data analysis or intimate knowledge of the algorithm.

This overlapping technique was not attempted on LSODE. The opportunities for exploiting this technique required heuristics to determine how often the calculation should be performed. The heuristics depended heavily upon the type and size of the ODE system being solved. Adding heuristics complicated the algorithm and would be difficult to implement. Since we had already received large gains from the previous work, the extra work was felt to be unjustified as only a small additional gain could be expected.

6.5. Comments on Conversion

The attempts to exploit parallelism internal to LSODE uncovered a few notable issues. First, the straightforward method was easy to implement, led to no problems, and produced a reasonably large overlap for the problem tested. However, it was felt that the cost of repetitive task creation and termination was excessive and was dealt with by pre-creating tasks once for each problem. This proved that the process was indeed costly because a third of a processor was gained by the pre-creation technique. However, the implementation of the pre-creation was very messy because of the lack of a correct data structure for the shared memory block. Also, Fortran did not provide a convenient method for passing information among tasks that are already created. These make the desirability of the pre-creation technique lower than it otherwise would have been.

The synchronization techniques used to signal tasks were easy to devise for these implementations. However, we did find it was very easy to produce mechanisms which did not work correctly. The majority of these led quickly to deadlock situations and abandonment of the mechanisms. The ability to use simple synchronization schemes increases the likelihood of automated concurrent code generation in the future.

Protecting the software package from the user, while exploiting parallelism dependent upon the user-supplied ODE function, proved to be hopeless. The user has complete control of the function. The only protection that can be supplied from within the domain of LSODE serializes the usage of the user-supplied ODE function, thus eliminating the desire to perform the work in the first place. Since no protection can be supplied, and still exploit the desired parallelism, the software package is at the mercy of the user. For example, supplying a non-reentrant ODE function may result in incorrect values returned to LSODE, which will cause an incorrect solution to be computed. Both LSODE and the user will be unaware of the incorrect answer. The extent of the damage caused by this can be enormous depending on the use to which the incorrect result is applied.

There were other techniques for exploiting parallelism that were not addressed in this work. First, we could replace the LINPACK system solving routines with their parallel versions. This should give us some immediate gain at little cost. Second, the overlapping DQJ technique could be used. Implementing this technique for LSODE requires both modification of the overall multistep algorithm and use of heuristics to determine how often a new DQJ should be calculated. An estimation of the cost of the heuristics compared to the cost of computing a new DQJ makes us believe that LSODE is not very amenable to this technique.

SECTION 7

Direct Results

We present a summary of the direct results obtained from each of the separate stages addressed the the preceding sections: *Data Analysis, Minimal Conversion Approach,* and *Internal Parallelism Approach.* This is merely a condensed refresher of the information already presented. In doing so, we emphasize those direct results which we believe are relevant to this work.

The *Data Analysis* helped identify concurrency problems and sections of code amenable to parallel processing. However, a complete data analysis was very difficult and time consuming. We found the intraprocedural analysis to be the easiest. Individual subroutines tend to have limited scope reducing the number of possible combinations for data dependencies. Large subroutines naturally tend to be more difficult. In either case, an enormous amount of information may be generated from a single routine increasing the burden of manual bookkeeping. The interprocedural analysis was by far the most difficult. If performed in a static manner, the number of possible interprocedural data dependencies may be large. If performed in a dynamic manner, the number of dependencies may be reduced at the expense of a more complicated determination of valid dependencies. The large number of data dependencies possible for even a small program make performing the data analysis manually very tedious and subject to error. Overlooking dependencies, already made difficult to determine by certain Fortran language constructs, may lead to days or even weeks of debugging a code which produces incorrect answers.

The tools available in our environment were helpful but insufficient. The CIVIC compiler listing provides the most information useful in performing the data analysis. Much of the intraprocedural analysis is already performed by the compiler. Yet, the majority of this information is not given to users. Only a small amount can be found in the compiler listing. However, without such information, performing a data analysis for this small software package would prove to be too burdensome to justify. The SC tool also had limited usefulness. Its single purpose was to help gain familiarity with the software package subroutine usage structure. The FOCAL tool supplies a very limited amount of useful information and was abandoned in favor of the CIVIC listing. Everything supplied by FOCAL is supplied by CIVIC and is presented in a more useful manner.

Limitations due to Fortran language constructs were found in the tools used. Many constructs inhibit the automatic detection of some intra- or interprocedural dependencies. The CIVIC compiler is unable to determine all intraprocedural dependencies because of pointered variables, equivalences, etc. While, the SC tool is unable to detect the call-tree structure associated with procedural parameters.

The *Minimal Conversion Approach* attempted to provide some insight into the problems and considerations associated with a quick and dirty conversion of an existing package to allow concurrent execution. A version of a software package usable in a multiprocessing environment may need to be supplied to satisfy short-term needs. We have tried to simulate this condition by imposing restrictions upon the work performed.

We found it difficult to maintain the syntax and semantics of the software during the conversion. Serial processing is a restricted form of parallel processing. Many of these restrictions are found to be implicit in the design of serial software. A few may be orthogonal to the purpose of a parallel package and inhibit its abilities. We found that attempts to maintain the interface and device management were cases that led to retention of the serial restrictions. Other imposed restrictions were mainly due to the lack of adequate language support to convert the serial data structures to ones which suitably handle the expanded flexibility of the parallel software.

The minimal conversion led to a severely limited, and hence unusable, software package. The combination of our self-imposed guidelines for the conversion and the problems mentioned above directed our decisions during the conversion. Thus, many restrictions were placed upon the parallel software produced. We felt these limitations were too great for the package to be usable. We were forced to abandon our guidelines and further the conversion until a version acceptable for usage was obtained.

The *Internal Parallelism Approach* expanded the previous work to examine the process of exploiting parallelism internal to the software package. This represents an attempt at a longer-term solution in supplying parallel software. The only guidelines placed upon this work were: (1) do not develop new parallel algorithms for the software but rather modify the existing internal algorithms to exploit parallelism found in them, and (2) only exploit areas where gains are likely.

Through the use of available tools, the program areas likely to be candidates for multiprocessing were easily found. Our guidelines allow us to identify execution hot-spots within serial code as candidates for exploiting parallelism. These areas are easy to find through use of the TIMER/TALLY tool. Once the candidate areas are defined, each area may be manually examined for possible exploitation of parallelism.

Problems encountered during this work made the exploitation more difficult than it needed to be. The lack of language support was the major contributor to the problems. These showed up as difficulty in sharing a data structure among a group of tasks and the inability to change a task's procedural parameter. Other problems came from the possible inappropriateness of the multitasking mechanism used for the granularity of the parallelism exploited. Also, the property of the software which displayed large amounts of candidate parallelism in its usage of the user-supplied function made protection of the function difficult.

SECTION 8

Conclusions

This paper has examined the process of converting an existing scientific software package to a multiple processor machine. The work was performed within the confines of a supercomputing environment, and a software package commonly used within this environment was used as a vehicle for this study. Thus, we were forced to use the tools and techniques currently available to our supercomputer users who may have to do similar work. The goal of this work has been to discover some of the difficulties which others may encounter, understand their cause, and discuss the options we found available for each.

We have found a need for better tools to aid in the process of converting software to multiple processor machines, particularly in the area of data analysis. The available tools supplied some aid to the analysis process, but information was limited. Even with the aid received, the data analysis process was very difficult and time consuming for the small software package converted. Clearly better tools will be required to perform similar work on large software packages. We would suggest an integrated set of tools (or a monolithic tool). It should be possible for one to sit at a terminal, start the tool, and inquire about various pieces of information concerning the software being analyzed. Types of information which must be supplied include: intra- and inter-procedural dependencies for any variable within or between specified code segments, the type and scope of any variable, and the ability to display the interrelationship of the procedures in the software package. This information must be supplied as accurately as possible within the limitations imposed by the langauge. Information which cannot be fully determined, for whatever reason, should be noted as such with a full explanation of the cause for failure so the user may follow it up manually. This will probably require that a database of information be formed (by the tool) for the software being analyzed. This will greatly decrease the bookkeeping which contributed to the difficulty of the manually performed analysis.

The syntax and semantics of serial packages must be reevaluated during the conversion to a multiple processor machine. Our attempt to maintain these during the conversion resulted in a package with too many limitations placed upon it to be considered usable. In addition, flexibility of usage allowed by a parallel environment may be inhibited by the implicit serial limitations of the original software.

Fortran lacks adequate language support for explicit concurrent programming. A large portion of the problems encountered throughout the work are attributable to the presence of constructs which do not fully coexist with the multitasking techniques supplied by the library of multitasking primitives. This in itself is not a problem with the language; rather the presence of the library represents a lack of support for multiprocessing within the language. The remaining problems were due to the inability to create data structures with the desired sharing. The data scopes supplied by the language do not fully encompass the task concept supplied by the library.

The performance obtained by each phase of the work was acceptable. A relatively low amount of work was necessary to produce near maximum benefits for the conversion allowing concurrent invocations. This is to be expected by definition of the problem. However, many application programs are unable to provide problems that may run concurrent with one another. The second phase attempted to exploit parallelism internal to each problem, thereby reducing the execution time for individual problems. This proved to be moderately successful. The straightforward method returned a large benefit, an overlap of 3.47 in the multiprocessed section, and required little modification to exploit it. However, the attempts to hide the overhead of the task creation primitive caused us problems due to lack of support for multitasking in the Fortran langauge. Nonetheless, an overlap of 3.77 was obtained by this method at the expense of a clean implementation.

An integrated analysis tool must be developed and made publicly available. The natural place for such a tool is as part of a compiler. Both the user and the compiler could benefit from its presence. Of course, a better approach is to have the compiler automatically generate concurrent code, thereby relieving the user from having to perform this work. Secondly, a comparison between automatic generation techniques (which do exist for other machines) and techniques similar to those used here should be made. An understanding of the limitations of automatic techniques should be made available to allow prospective users the ability to intelligently select the technique appropriate to their needs. Finally, we must understand the language requirements necessary to adequately support explicit concurrent programming.

Acknowledgements

I would like to express my appreciation to Dr. Roger White, Dr. Robert Haight and numerous others of the Experimental Physics division at Lawrence Livermore National Laboratory (LLNL) for their continued support and encouragement throughout my academic career. I wish to thank Dr. Steve Skedzielewski and Dr. Alan Hindmarsh of the Computing and Mathematics Research Division at LLNL for their criticisms, advice and suggestions towards this thesis. A special thanks goes to Alan Hindmarsh for contributing much of his time in consultation and enlightenment on the technical aspects of LSODE, and for supplying the initial nearly reentrant version of the LSODE package. Finally, I am indebted to my thesis advisor, Dr. James McGraw, for his time, encouragement, and suggestions. His devotion to this work helped provide me with the motivation necessary to overcome outside pressures and complete this work. His guidance has been invaluable in keeping me from straying from the true subject of this thesis. It has been my pleasure to have been able to work with him.

Appendix A

The program segments which make up each of the four implementations discussed in Section 6, **Internal Parallelism Approach**, are listed here. They are listed to supply further detail for the interested reader.

Straightforward

• *coding used at the point of the original loop 230 in the Jacobian formation:*

```
c  divide up the work between the tasks.
      numiter = n/NTASKS
      do 220 i = 1,NTASKS
         mybeg(i) = (i-1)*numiter+1
         myend(i) = i*numiter
         tinfo(1,i)  = 2
  220 continue
      myend(NTASKS) = n
c
c  start up ntasks-1 new tasks and then do the remaining work locally.
      do 224 i = 1,(NTASKS-1)
         call tskstart ( tinfo(1,i), loop230, mybeg(i), myend(i), f, y, savf, ewt, wm, neq, n,
     x                   tn, r0, hl0 )
  224 continue
      call loop230 ( mybeg(NTASKS), myend(NTASKS), f, y, savf, ewt, wm, neq, n,
     x               tn, r0, hl0 )
c
c  now synchronize until everyone has finished.
      do 226 i = 1,(NTASKS-1)
         call tskwait (tinfo(1,i))
  226 continue
```

• *the relocated code for loop 230 which now forms the taskhead:*

```
      subroutine loop230 ( ibeg, iend, f, yy, savf, ewt, wm, neq, n, tn, r0, hl0 )
      external  f
      integer  ibeg, iend, n, neq
      integer  i, j, j1
      real  ewt, hl0, r0, savf, tn, wm, yy
      real  fac, ftem, r, srur, y, yj
      dimension  yy(1), savf(1), ewt(1), wm(1), neq(1)
      pointer  (qy, y(1))
      pointer  (qftem, ftem(1))
c
c  allocate local copies of the 'yy' array and 'ftem' workspace
      call mzalloc ( qy, 2*n )
      qftem = qy+n
      do 100 i = 1,n
  100    y(i) = yy(i)
      srur = wm(1)
      j1  = 2+(ibeg-1)*n
c
c  release the allocated space
      call mzdalloc ( qy, 2*n )
      return
      end
```

Pre-create with Barrier

- *the pre-creation coding used at the beginning of a problem:*

```
c  allocate the work spaces and divide up the work
       if (mf .ne. 22) go to 109
           nmpwork = (5*NTASKS) + (NTASKS*n*2)  + 16
           call mzalloc ( qmpwork, nmpwork )
           mparea = qmpwork
           qirdylok = qmpwork+12
           qideclok = qirdylok+1
           qirdywrk = qideclok+1
           qidonwrk = qirdywrk+1
           qmybeg = qidonwrk+1
           qmyend = qmybeg+NTASKS
           qtinfo = qmyend+NTASKS
           qmywloc = qtinfo+(2*NTASKS)
           nmpwork = qmywloc+NTASKS
           numiter = n/NTASKS
           do 101 i = 1,NTASKS
               mybeg(i) = (i-1)*numiter+1
               myend(i) = i*numiter
               mywloc(i) = nmpwork+(i-1)*n*2
               tinfo(1,i) = 2
 101       continue
           myend(NTASKS) = n
c
c  initialize the sync and pre-create the tasks
           call barasgn ( irdywrk, NTASKS+1 )
           call evasgn ( idonwrk )
           call lockasgn ( irdylok )
           call lockasgn ( ideclok )
           do 102 i = 1,NTASKS
               call tskstart ( tinfo(1,i), loop230, mpwork(1), mybeg(i), myend(i), f,
   x                          irdylok, ideclok, irdywrk, idonwrk, mywloc(i) )
 102       continue
 109   continue
```

- *the coding used at the point of the original loop 230 in the Jacobian formation:*

```
c  set up the addresses to the information to act as a secondary argument list.
       jcount = NTASKS
       qmpwork = mparea
       qirdywrk = qmpwork+14
       qidonwrk = qmpwork+15
       mpwork(1) = .loc.y
       mpwork(2) = .loc.savf
       mpwork(3) = .loc.ewt
       mpwork(4) = .loc.wm
       mpwork(5) = .loc.neq
```

```
      mpwork(6) = .loc.n
      mpwork(7) = .loc.tn
      mpwork(8) = .loc.r0
      mpwork(9) = .loc.hl0
      mpwork(10) = .loc.jcount
c
c  signal the tasks that we are set up and ready to go.
c  and wait for them to tell us of their completion.
      call barsync ( irdywrk )
      call evwait ( idonwrk )
      call evclear ( idonwrk )
```

- *the relocated code for loop 230 which now forms the taskhead:*

```
      subroutine loop230 ( mpwork,ibeg,iend,f,irdylok,ideclok,irdywrk,idonwrk,locwork )
      external f
      integer ibeg, iend, ideclok, idonwrk, irdylok, irdywrk, locwork, mpwork
      integer i, j, j1, n, neq
      real ewt, fac, ftem, hl0, r, r0, savf, srur, tn, wm, y, yj, yy
      dimension mpwork(1)
      pointer ( qyy, yy(1) ), ( qsavf, savf(1) ), ( qewt, ewt(1) ), ( qwm, wm(1) )
      pointer ( qneq, neq(1) ), ( qn, n ), ( qtn, tn ), (qr0, r0 ), (qhl0, hl0 )
      pointer ( qjcount, jcount ), ( qy, y(1) ), ( qftem, ftem(1) )
c
c  wait until all tasks (including parent) signal they are ready
 10   call barsync ( irdywrk )
c
c  pick the shared data (arg-list) and make a local copy of y.
      qyy = mpwork(1)
      qsavf = mpwork(2)
      qewt = mpwork(3)
      qwm = mpwork(4)
      qneq = mpwork(5)
      qn = mpwork(6)
      qtn = mpwork(7)
      qr0 = mpwork(8)
      qhl0 = mpwork(9)
      qjcount = mpwork(10)
      qy = locwork
      qftem = qy+n
      do 100 i = 1,n
 100     y(i) = yy(i)
c
      j1 = 2+(ibeg-1)*n
      srur = wm(1)
      do 230 j = ibeg,iend
         yj = y(j)
         r = amax1( srur*abs(yj), r0/ewt(j) )
         y(j) = y(j)+r
         fac = -hl0/r
```

```
            call f ( neq, tn, y, ftem )
            do 220 i = 1,n
  220           wm(i+j1) = (ftem(i)-savf(i))*fac
            y(j) = yj
            j1 = j1+n
  230   continue
c
c   last task to complete work will signal the parent
        call lockon ( ideclok )
            jcount = jcount-1
            if (jcount .eq. 0) call evpost ( idonwrk )
        call lockoff ( ideclok )
        go to 10
        end
```

Pre-create with Barrier and Self Scheduling

- *the pre-creation coding used at the beginning of a problem:*

```
c  allocate the work spaces and divide up the work
      if (mf .ne. 22) go to 109
          nmpwork = (3*NTASKS) + (NTASKS*n*2)  + 16
          call mzalloc ( qmpwork, nmpwork )
          mparea = qmpwork
          qirdylok = qmpwork+12
          qideclok = qirdylok+1
          qirdywrk = qideclok+1
          qidonwrk = qirdywrk+1
          qtinfo = qidonwrk+1
          qmywloc = qtinfo+(2*NTASKS)
          nmpwork = qmywloc+NTASKS
          do 101 i = 1,NTASKS
              mywloc(i) = nmpwork+(i-1)*n*2
              tinfo(1,i) = 2
 101      continue
c
c  initialize the sync and pre-create the tasks
          call barasgn ( irdywrk, NTASKS+1 )
          call evasgn ( idonwrk )
          call lockasgn ( irdylok )
          call lockasgn ( ideclok )
          do 102 i = 1,NTASKS
              call tskstart ( tinfo(1,i), loop230, mpwork(1), f,
     x                     irdylok, ideclok, irdywrk, idonwrk, mywloc(i) )
 102      continue
 109  continue
```

- *the coding used at the point of the original loop 230 in the Jacobian formation:*

```
c  set up the loop control variables
      nextj = 1
      maxj = n
      jcount = maxj-nextj+1
c  set up the addresses to the information to act as a secondary argument list.
      qmpwork = mparea
      qirdywrk = qmpwork+14
      qidonwrk = qmpwork+15
      mpwork(1) = .loc.y
      mpwork(2) = .loc.savf
      mpwork(3) = .loc.ewt
      mpwork(4) = .loc.wm
      mpwork(5) = .loc.neq
      mpwork(6) = .loc.n
      mpwork(7) = .loc.tn
      mpwork(8) = .loc.r0
      mpwork(9) = .loc.hl0
      mpwork(10) = .loc.nextj
      mpwork(11) = .loc.maxj
      mpwork(12) = .loc.jcount
c
c  signal the tasks that we are set up and ready to go.
c  and wait for them to tell us of their completion.
      call barsync ( irdywrk )
      call evwait ( idonwrk )
      call evclear ( idonwrk )
```

• *the relocated code for loop 230 which now forms the taskhead:*

```
      subroutine loop230 (mpwork,f,irdylok,ideclok,irdywrk,idonwrk,locwork )
      external  f
      integer  ibeg, iend, ideclok, idonwrk, irdylok, irdywrk, locwork, mpwork
      integer  i, j, j1,jcount, maxj, nextj, n, neq
      real  ewt, fac, ftem, hl0, r, r0, savf, srur, tn, wm, y, yj, yy
      dimension mpwork(1)
      pointer ( qyy, yy(1) ), ( qsavf, savf(1) ), ( qewt, ewt(1) ), ( qwm, wm(1) )
      pointer ( qneq, neq(1) ), ( qn, n ), ( qtn, tn ), (qr0, r0 ), (qhl0, hl0 )
      pointer ( qjcount, jcount ), ( qy, y(1) ), ( qftem, ftem(1) )
      pointer ( qnextj, nextj ), (qmaxj, maxj )
c
c  wait until all tasks (including parent) signal they are ready.  On initialization, avoid clearing
c  the lock...Otherwise the task signaling completion must clear the lock before synching)
      go to 10
  5   call lockoff ( ideclok )
 10   call barsync ( irdywrk )
c
c  pick the shared data (arg-list) and make a local copy of y.
      qyy = mpwork(1)
      qsavf = mpwork(2)
      qewt = mpwork(3)
      qwm = mpwork(4)
      qneq = mpwork(5)
      qn = mpwork(6)
      qtn = mpwork(7)
      qr0 = mpwork(8)
      qhl0 = mpwork(9)
      qnextj = mpwork(10)
      qmaxj = mpwork(11)
      qjcount = mpwork(12)
      qy = locwork
      qftem = qy+n
      do 100 i = 1,n
 100      y(i) = yy(i)
c
      srur = wm(1)
c
```

```
c  have tasks pick up a single iteration (self-scheduling loop).  if all the work is claimed,
c  send the task to the completion synch point.
 200  continue
      call lockon ( irdylok )
           j = nextj
           nextj = nextj+1
      call lockoff ( irdylok )
      if (j .gt. maxj ) go to 10
c
           j1 = 2+(j-1)*n
           yj = y(j)
           r = amax1( srur*abs(yj), r0/ewt(j) )
           y(j) = y(j)+r
           fac = -hl0/r
           call f ( neq, tn, y, ftem )
           do 220 i = 1,n
 220            wm(i+j1) = (ftem(i)-savf(i))*fac
           y(j) = yj
c
c  last task to complete work will signal the parent
      call lockon ( ideclok )
           jcount = jcount-1
           if (jcount .ne. 0) go to 300
                call evpost ( idonwrk )
                go to 5
 300       continue
      call lockoff ( ideclok )
      go to 200
      end
```

Pre-create with Events and Self Scheduling

- *the pre-creation coding used at the beginning of a problem:*

```
c   allocate the work spaces and divide up the work
          if (mf .ne. 22) go to 109
              nmpwork = (4*NTASKS) + (NTASKS*n*2)  + 15
              call mzalloc ( qmpwork, nmpwork )
              mparea = qmpwork
              qirdylok = qmpwork+12
              qideclok = qirdylok+1
              qirdywrk = qideclok+1
              qidonwrk = qirdywrk+NTASKS
              qtinfo = qidonwrk+1
              qmywloc = qtinfo+(2*NTASKS)
              nmpwork = qmywloc+NTASKS
              do 101 i = 1,NTASKS
                  mywloc(i) = nmpwork+(i-1)*n*2
                  tinfo(1,i) = 2
                  call evasgn ( irdywork(i) )
  101         continue
c
c   initialize the sync and pre-create the tasks
              call evasgn ( idonwrk )
              call lockasgn ( irdylok )
              call lockasgn ( ideclok )
              do 102 i = 1,NTASKS
                  call tskstart ( tinfo(1,i), loop230, mpwork(1), f,
     x                         irdylok, ideclok, irdywrk(i), idonwrk, mywloc(i) )
  102         continue
  109   continue
```

- *the coding used at the point of the original loop 230 in the Jacobian formation:*

```
c   set up the loop control variables
        nextj = 1
        maxj = n
        jcount = maxj-nextj+1
c   set up the addresses to the information to act as a secondary argument list.
        qmpwork = mparea
        qirdywrk = qmpwork+14
        qidonwrk = qmpwork+14 + NTASKS
        mpwork(1) = .loc.y
        mpwork(2) = .loc.savf
        mpwork(3) = .loc.ewt
        mpwork(4) = .loc.wm
        mpwork(5) = .loc.neq
        mpwork(6) = .loc.n
        mpwork(7) = .loc.tn
        mpwork(8) = .loc.r0
        mpwork(9) = .loc.hl0
        mpwork(10) = .loc.nextj
        mpwork(11) = .loc.maxj
        mpwork(12) = .loc.jcount
c
c  signal the tasks that we are set up and ready to go.
c  and wait for them to tell us of their completion.
        do 239 i = 1,NTASKS
             call evpost ( irdywrk(i) )
  239   continue
        call evwait ( idonwrk )
        call evclear ( idonwrk )
```

• *the relocated code for loop 230 which now forms the taskhead:*

```
        subroutine loop230 (mpwork,f,irdylok,ideclok,irdywrk,idonwrk,locwork )
        external  f
        integer  ibeg, iend, ideclok, idonwrk, irdylok, irdywrk, locwork, mpwork
        integer  i, j, j1,jcount, maxj, nextj, n, neq
        real  ewt, fac, ftem, hl0, r, r0, savf, srur, tn, wm, y, yj, yy
        dimension mpwork(1)
        pointer ( qyy, yy(1) ), ( qsavf, savf(1) ), ( qewt, ewt(1) ), ( qwm, wm(1) )
        pointer ( qneq, neq(1) ), ( qn, n ), ( qtn, tn ), (qr0, r0 ), (qhl0, hl0 )
        pointer ( qjcount, jcount ), ( qy, y(1) ), ( qftem, ftem(1) )
        pointer ( qnextj, nextj ), (qmaxj, maxj )
c
c  wait until all tasks (including parent) signal they are ready.  On initialization, avoid clearing
c  the lock...Otherwise the task signaling completion must clear the lock before synching)
        go to 10
  5     call lockoff ( ideclok )
  10    call evwait ( irdywrk )
        call evclear ( irdywrk )
c
c  pick the shared data (arg-list) and make a local copy of y.
        qyy = mpwork(1)
        qsavf = mpwork(2)
        qewt = mpwork(3)
        qwm = mpwork(4)
        qneq = mpwork(5)
        qn = mpwork(6)
        qtn = mpwork(7)
        qr0 = mpwork(8)
        qhl0 = mpwork(9)
        qnextj = mpwork(10)
        qmaxj = mpwork(11)
        qjcount = mpwork(12)
        qy = locwork
        qftem = qy+n
        do 100 i = 1,n
  100        y(i) = yy(i)
```

```
c
      srur = wm(1)
c
c  have tasks pick up a single iteration (self-scheduling loop).  if all the work is claimed,
c  send the task to the completion synch point.
  200 continue
      call lockon ( irdylok )
         j = nextj
         nextj = nextj+1
      call lockoff ( irdylok )
      if (j .gt. maxj ) go to 10
c
         j1 = 2+(j-1)*n
         yj = y(j)
         r = amax1( srur*abs(yj), r0/ewt(j) )
         y(j) = y(j)+r
         fac = -hl0/r
         call f ( neq, tn, y, ftem )
         do 220 i = 1,n
  220       wm(i+j1) = (ftem(i)-savf(i))*fac
         y(j) = yj
c
c  last task to complete work will signal the parent
      call lockon ( ideclok )
         jcount = jcount-1
         if (jcount .ne. 0) go to 300
            call evpost ( idonwrk )
            go to 5
  300    continue
      call lockoff ( ideclok )
      go to 200
      end
```

References

[ABKP 86] R. Allen, D. Bäumgartner, K. Kennedy and A. Porterfield, "PTOOL: A Semi-automatic Parallel Programming Assistant," Technical Report COMP TR86-31, Rice University (January 1986).

[Alla 85] S.J. Allan and R.R. Oldehoeft, "HEP SISAL: Parallel Functional Programming," in *Parallel MIMD Computation: The HEP Supercomputrer and its Applications*, J.S. Kowalik, ed., MIT Press, Cambridge, Mass. (1985) pp. 123-150.

[ApMc 85] W.F. Appelbe and C.E. McDowell, "Anomaly Reporting - a Tool for Debugging and Developing Parallel Numerical Algorithms," *Proc. IEEE 1st Int. Conf. on Supercomputing Systems,* (December 1985) pp. 386-391.

[Brin 75] P. Brinch-Hansen, "The Programming Langauge Concurrent Pascal," *IEEE Trans. on Software Engineering* (June 1975) pp. 199-207.

[ChDH 84] S.C. Chen, J.J. Dongarra and C.C. Hsiung, "Multiprocessing Linear Algebra Algorithms on the Cray X-MP-2: Experiences with Small Granularity," Technical Memorandum No. 24, Mathematics and Computer Science Division, Argonne National Laboratory (February 1984).

[Coop 82] R.E. Cooper, "FOCAL," LCSD-1630, Lawrence Livermore National Laboratory (September 1982).

[Cray 86] Cray Research Inc., "Fortran (CFT) Reference Manual," SR-0009 Revision L (February 1986) pp. 4.24-4.27.

[Cray 85] Cray Research Inc., "Multitasking User Guide," Technical Note SN-0222 Revision A (January 1985).

[Cray 84] Cray Research Inc., "Cray X-MP Series Model 48 Mainframe Reference Manual," HR-0097 (August 1984).

[Cray 80] Cray Research Inc., "Cray 1 S Hardware Reference Manual," HR-080S Revision 02 (June 1980).

[DBMS 79] J.J. Dongarra, J.R. Bunch, C.B. Moler and G.W. Stewart, *LINPACK User's Guide*, SIAM, Philadelphia (1979).

[DeEM 82] W.S. Derby, J.T. Engle and J.T. Martin, "Lrltran Language Used with the CHAT and CIVIC Compilers," LCSD-302 Revision 1, Lawrence Livermore National Laboratory (June 1982).

[Deit 83] H.M. Deitel, *An Introduction to Operating Systems*, Addison-Wesley, Reading, Mass. (1983) pp. 75-97

[DiKl 85] H. Dietz and D. Klappholz, "Refined C: A Sequential Language for Parallel Programming," *Proc. IEEE 1985 Int. Conf. on Parallel Processing* (August 1985) pp. 442-449.

[DuBo 84] P.J. DuBois, "Development of the NLTSS Operating System," UCRL-90923, Lawrence Livermore National Laboratory (June 1984).

[FrJS 85] P.O. Fredrickson, R.E. Jones and B.T. Smith, "Synchronization and Control of Parallel Algorithms," *Parallel Computing* **2**,3 (November 1985) p. 255.

[Gear 71] C.W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ (1971) pp. 158-166.

[Hind 82] A.C. Hindmarsh, "ODEPACK, A Systematized Collection of ODE Solvers," *Scientific Computing*, R.S. Stepleman, eds., North-Holland, Amsterdam (1983) pp. 55-64.

[Hind 72] A.C. Hindmarsh, "Linear Multistep Methods for Ordinary Differential Equations: Method Formulations, Stability, and the Methods of Nordsieck and Gear," UCRL-51186 Revision 1, Lawrence Livermore National Laboratory (March 1972).

[Horn 75] R.W. Hornbeck, *Numerical Methods*, Quantum, New York, NY (1975) pp. 185-203.

[Jone 78] R.E. Jones, "SLATEC Common Mathematical Library Error Handling Package," SAND78-1189, Sandia National Laboratories (September 1978).

[Kuck 84] D.J. Kuck et al., "The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance," *Proc. IEEE 1984 Int. Conf. on Parallel Processing* (August 1984) pp. 129-138.

[KKPL 81] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Proc. 8th ACM Symp. on Principles of Programming Languages* (January 1981) pp. 207-218.

[Lars 84] J.L. Larson, "Multitasking on the Cray X-MP-2 Multiprocessor," *IEEE Computer* **17**,7 (July 1984) pp. 62-69.

[McAx 84] J.R. McGraw and T.S. Axelrod, "Exploiting Multiprocessors: Issues and Options," UCRL-91734, Lawrence Livermore National Laboratory (October 1984).

[McGr 83] J.R. McGraw, et al., "SISAL: Streams and Iteration in a Single-Assignment Language; Language Reference Manual," Report M-146, Lawrence Livermore National Laboratory (July 1983).

[McKW 84] J.R. McGraw, D.J. Kuck and M.Wolfe, "A Debate: Retire FORTRAN?," *Physics Today* **37,5** (May 1984) pp. 66-75.

[Nels 81a] B.J. Nelson, "Remote Procedure Call," CSL-81-9, Xerox Palo Alto Research Center (May 1981) pp. 162-163.

[Nels 81b] H.L. Nelson, "Timing Codes on the Cray-1: Principles and Applications," UCID-30179 Revision 2, Lawrence Livermore National Laboratory (May 1981) pp. 2-5.

[OHai 82] K. O'Hair, "SC," Internal Document, Lawrence Livermore National Laboratory (1982).

[Zimm 85] D.L. Zimmerman, "An Examination of Programming Support Environments," U.C. Davis/Livermore (December 1985) Qualifying Exam Paper.